ENHANCING CAPABILITIES OF THE NETWORK DATA PLANE USING NETWORK VIRTUALIZATION AND SOFTWARE DEFINED NETWORKING

A Thesis
Presented to
The Academic Faculty

by

Muhammad Bilal Anwer

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2015

ENHANCING CAPABILITIES OF THE NETWORK DATA PLANE USING NETWORK VIRTUALIZATION AND SOFTWARE DEFINED NETWORKING

Approved by:

Professor Nicholas G. Feamster,
Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ling Liu
Scool of Computer Science
Georgia Institute of Technology

Professor Mostafa H. Ammar
School of Computer Science
Georgia Institute of Technology

Professor Theophilus Benson
Department of Computer Science
Duke University

Professor Ellen W. Zegura
Scool of Computer Science
Georgia Institute of Technology

Date Approved: 30 Oct 2015

To my family

ACKNOWLEDGEMENTS

I am immensely grateful to my PhD adviser Prof. Nick Feamster. Throughout my PhD journey his guidance, support and supervision with a friendly attitude is something that has helped me keep going in this program. Nick's demand for higher quality combined with his support has made this dissertation possible.

I would like to thank my co-authors on my research publications who have helped me during various stages of different projects. I would specially like to thank Murtaza Motiwala, Mukarram Bin Tariq, Ankur Nayak, Prof. Ling Liu, Sam Burnett, Dave Levin, Theophilus Benson, Prof. Jennifer Rexford for being patient with me and giving me the opportunity to work with them and learn from them.

I am grateful to my thesis committee members who were patient with me and helped me polish my dissertation and bring it to its current state. I would like to thank Prof. Ellen Zegura and Prof. Mostafa Ammar for their kind advice during my stay at Georgia Tech. I would like to thank Prof. Ling Liu for her continuous encouragement and support. Her insightful questions combined with confidence in me are something that were instrumental in building part of this thesis. I am also very grateful to Prof. Theophilus Benson who supported me and encouraged me to finish the last mile of my PhD dissertation.

I would like to thank my NTG(Networking and Telecommunications Group) colleagues, specially Aemen H. Lodhi, Yogesh Mundada, Ahmed Mansy, Karim Habak, Maria Konte, and Samantha Lu for their support for both my PhD and non-PhD life.

I am also thankful to my friends Muhammad Raza Khan, Qasim Javed, Moazzam Khan and Mohammad Omer. They helped me both in the school and outside the school and were more than happy to lend an ear when life got tough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Enhancement of network data-plane functionality is an open problem that has recently gained momentum. Addition and programmability of new functions inside the network data-plane to enable high speed, complex network functions with minimum resource utilization, is main focus of this thesis. In this work, we look at different levels of the network data-plane design and using network virtualization and software defined networking we propose data-plane enhancements to achieve these goals. This thesis is divided into two parts, in first part we take a ground up approach where we focus our attention at the fast path packet processing. Using hardware and software based network virtualization we show how hardware and software based network switches can be designed to achieve above mentioned goals. We then present a switch design to quickly add these custom fast path packet processors to the network data-plane using software defined networking. In second part of this thesis we take a top to bottom approach where we present a programming abstraction for network operators and a network function deployment system for this programming abstraction. We use network virtualization and software defined networking to introduce new functions inside the network data-plane while alleviating the network operators of the deployment details and minimizing the network resource utilization.

CHAPTER 1

INTRODUCTION

Traditional networks mainly consist of end hosts, packet forwarding devices and the connecting medium(wired or wireless). This dissertation deals with research work done in the area of packet forwarding devices and their programmability. Packet forwarding devices include switches, routers and middleboxes etc. With the improvement in technology the boundaries between different packet forwarding devices are blurring. In this dissertation we focus our attention on packet forwarding devices that generally fall into switches, router and middleboxes categories.

Functionality of packet forwarding devices can be divided into control and data-plane. Data plane is responsible for forwarding the packets and control plane is responsible for establishing, maintaining and populating the forwarding tables of the data plane device. Traditional packet forwarding devices have the control and data plane packaged into a single device. The control plane runs on the same device as the data plane. This allows each packet forwarding device to act independently from other devices inside the network. This approach has worked for the networking community for decades but with increasing network sizes and increasing configuration complexity it has become more difficult to manage these networks [93] and has made them more prone to errors [28].

Apart from difficulty in management and increase in errors, vertically integrated packet forwarding devices have more importantly contributed in slow progress in network control and data plane innovations. The standardization process to introduce new changes in the protocols is lengthy and painful. Furthermore, any changes or

modifications in the devices supporting new protocol is a long process requiring convergence of standardization committee and support from network equipment vendors.

In part, limitations to introduce any changes in the packet forwarding devices gradually pushed networking community towards centralization and separation of control and data plane [37, 41, 74, 151, 164]. This separation and centralization ultimately paved way for innovations in the control plane and revival of software defined networking.

Separation of control plane and data plane allows the network operators to easily introduce new control plane protocols. Before software defined networking any innovation at the control plane required the approval and support from the network equipment vendor. Separation and centralization of control plane [57, 117, 145, 164] has allowed network operators to introduce new control plane features without any support from network equipment vendors. Apart from custom control plane protocols SDN(Software Defined Networking) also allows network operators to run multiple control planes in parallel [139].

Control plane innovations have enabled network operators to introduce new protocols into their network without any support from the network vendors [139]. Even with this centralization and separation, the innovation towards data plane functions has largely been dependent on the standardization process from the standardization committees. Most significant efforts in this regard have been led by Open Networking Foundation [115] and IETF(Internet Engineering Task Force) [84]. Standardization efforts though necessary can result in very slow evolution process for network data-plane. From perspective of data plane evolution, this situation brings us to the same point in networking history (*i.e.*before SDN came into limelight and before Active Networking [65]), when network community had to wait for standardization process and network equipment vendors to update any control plane protocols.

Network Virtualization: In this thesis we show how data-plane innovations

can be introduced without any help from network data-plane equipment vendor. One of the first requirement to introduce custom data planes in an existing network is to be able to add new functions alongside existing functions. This is where we exploit network virtualization [26, 49, 50, 104, 152] and show how data plane functions can be run in parallel with existing network data planes in chapters 3, 4 and 5.

Software Defined Networking: Introduction of data plane enhancements inside an existing virtualized data plane requires a mechanism that can be used to add new data plane changes after development or remove them once they are not required. In this work we use software defined networking as a tool to manage the data plane functions and add or remove them using a programming abstraction(chapters 5, 6 and 7).

Enhancement of data plane functions requires network virtualization to run data planes in parallel and software defined networking is used to manage these functions inside the data plane. But apart from parallel data planes and management of network functionality, this dissertation also focuses on basic requirements of data plane functions which include but are not limited to forwarding speed, function complexity and resource utilization. In this dissertation, we show how these requirements for programmable data planes can be met using network virtualization, software defined networking and a combination of multiple hardware/software technologies.

1.1 Network Planes

Network data plane consists of multiple parts, it includes routers, switches, middleboxes and many other packet forwarding and signal enhancing devices. Here we focus our attention towards enhancing functionality of packet forwarding devices and middleboxes/network functions.

Traditionally networks are logically divided into control plane and data plane. In context of SDN, control plane is physically separate from data plane. Control

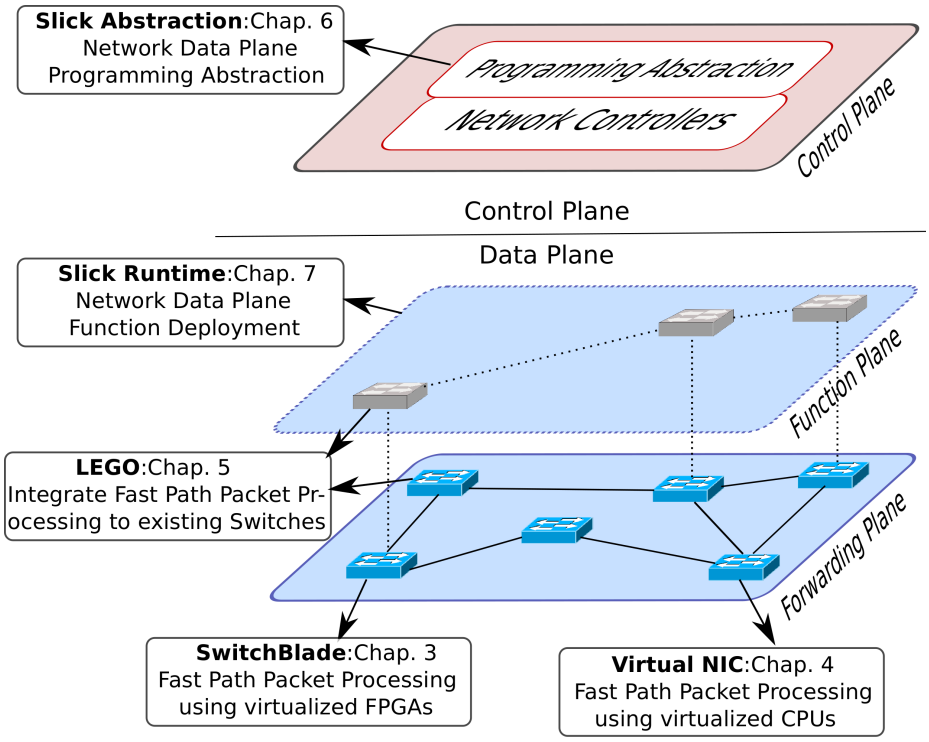


Figure 1: Dissertation Contributions

plane resides on logically central node and data plane is supported by multiple packet forwarding devices distributed across the network [108]. In this dissertation we further divide network data plane into two parts: *Function Plane* and *Forwarding Plane* as shown in figure 1. Forwarding Plane devices are responsible for forwarding the packets from source to destination. Without these devices the end to end network connectivity is not possible. Examples of such devices include but are not limited to switches, routers, NAT(Network Address Translation) [86] devices etc. These devices constitute *Forwarding Plane* of the data-plane as shown in figure 1. The second category of devices is responsible for processing the packets but in absence of these devices, packet reachability is not affected but correctness of network policy is affected. Majority of middleboxes or virtual network functions are included in this category. These devices include but are not limited to Firewalls, Load Balancers, Intrusion Detection System, Intrusion Prevention Systems etc. These devices constitute *Function Plane* of network data-plane. Here we note that this separation of Forwarding and Function plane is

not physical but logical. It is possible that a Function Plane *device* is on the same physical machine as a Forwarding Plane *device* (Heterogeneous Switch in Chapter 5) as shown in figure 1.

In this dissertation we show how virtualization, software defined networking and a combination of different semiconductor technologies can be used to provide manageable, high speed, complex network functions with minimal resource utilization of network data plane.

1.2 Thesis Contributions

Figure 1 shows contributions of this thesis at different levels of a network data plane. Our contributions range from fitting multiple network functions on hardware chip at line speed to running complex network functions inside the network data-plane, while striving for minimum resource utilization and high packet forwarding speeds.

In this dissertation we present contributions at different levels of network planes(Figure 1). We present a programmable virtualized fast path packet processing architecture to run multiple forwarding plane protocols side by side on FPGA(Field Programmable Gate Array) in chapter 3. Then we show how a virtualized software based data plane device architecture can be enhanced with support from hardware modifications in chapter 4. These chapters make contributions in fast path packet processors of network data plane as shown in figure 1. Chapter 5 builds a packet forwarding device using lessons learned in previous chapters and shows how the enhancements made in previous chapters can be integrated to an existing packet forwarding device in a network. Chapter 6 develops a programming abstraction for controller applications and network functions along with a number of network functions that can be used to add custom network functionality on top of network forwarding plane. With enhanced devices in network forwarding and function plane the questions arise about how to place enhanced devices/new functions inside the network and steer traffic through

them while minimizing the end to end latency and end to end bandwidth utilization of overall network resources. Chapter 7 presents a deployment system with multiple algorithms that can be used to minimize the overall network resource utilization while deploying new network functions as shown in figure 1.

1.3 Road Map

Following is the organization of rest of this thesis and contributions made in each of its chapters.

- In chapter 3 we show how network hardware virtualization can be used to add custom data plane functions and how these functions can be run at line rate side by side. For this we design, implement and evaluate an architecture called SwitchBlade.

- In chapter 4 we show how software based virtualized data-plane device performance can be improved with small changes in hardware of network interface cards.

- Based on the lessons and experience learned in chapters 3 and 4 we show how heterogeneous hardware components can be used to enhance data plane functionality using network virtualization and SDN, in chapter 5.

- In chapter 6 we present a programming abstraction that can be used to introduce new functions inside a network. This abstraction requires changes at control plane, data plane and the protocol level.

- Lastly in chapter 7 we present design and implementation of a runtime system that can be used to introduce functions at the data-plane level resulting in lower latencies and lower end to end bandwidth utilization for network policies.

- We finish this dissertation with concluding remarks in chapter 8. We present some of the possible directions that can be taken to further enhance the network data plane capabilities.

CHAPTER 2

BACKGROUND AND RELATED WORK

Network data planes have been under research and development since the inception of packet forwarding devices [81]. Commercialization and wide adaption of network switches and routers increased the need for high speed packet forwarding. High packet processing requirements combined with the need for performance and resource utilization resulted in reduced focus on forwarding device programmability by the network equipment vendor community.

Before the wide adoption of software defined networking, packet forwarding devices had their control and data plane inside a single device. The technologies used to implement control plane in these devices were low cost embedded processors for control plane functions and ASICs(Application Specific Integrated Circuits), FPGAs(Field Programmable Gate Arrays), Network Processors for packet forwarding in fast path.

With separation of control plane and data plane in SDN(Software Defined Networks), general purpose processors are used for control plane programming and different technologies have been used for data plane functions. There has been a lot of work to enhance data plane functionality for both SDN and non-SDN packet forwarding devices. Broadly speaking data-plane functions can be implemented using multiple hardware technologies. These technologies include but are not limited to ASICs(Application Specific Integrated Circuits), FPGAs(Field Programmable Gate Arrays), CPUs(Central Processing Units), GPUs(Graphics Processing Units) and many other technologies using a combination of the above technologies.

Enhancement of data plane can either be done by replacing old data plane hardware functions with new ones after testing them in lab for limited time period. Another approach is to run old and new protocols side by side and gradually move the traffic from old protocols to new protocols while iterating on the design and development of new data plane functions. In this dissertation we follow the second approach where new data plane functions can be introduced side by side with old data plane functionalities.

In this chapter we discuss work previously presented in networking literature and talk about how the contributions made in this dissertation can be placed in the context of work presented by other researchers. We present work done by various researchers in the area of programmable data planes and show how virtualization, software defined networking and different hardware technologies are used to enhance functionality of data planes and how our contributions fit into this body of research work.

In this thesis, the question of network data plane function enhancements is divided into two main questions: 1) How to enhance the data plane device's fast path processors(FPGA and CPU) and integrate them in an existing network data plane? 2) How to program the network functions inside the whole network? To answer these questions we take two approaches. To answer first question we take ground up approach in first part of thesis(chapters 3,4,5) where we enhance functionality of FPGA and CPU based fast path packet processors and integrate them using LEGO architecture (chapter 5) in network data plane. The second part of this dissertation tackles the second question in chapters 6,7 by developing a programming abstraction and developing a network function deployment system, respectively.

2.1 Ground Up Approach

Running new data plane functions with old functions ,side by side, requires the data plane devices to provide the high speed and complex functions with minimal packet forwarding device resource utilization. In this dissertation we show how packet forwarding devices can be enhanced to run multiple protocols side by side using network virtualization in chapters 3,4,5. In section 2.1.1 we present the research work related to SwitchBlade(Chap. 3) that shows how multiple protocols can be run side by side with high packet forwarding speeds and better programmability. In section 2.1.2 we present related research work done in the area of improving performance of CPU virtualization that can be used to host virtual machines for running applications as well as packet forwarding functions in data plane devices. Then in section 2.1.3 we present research work related to our work on LEGO, an architecture to integrate enhanced packet forwarding devices in existing data plane devices.

2.1.1 Virtualized Hardware Packet Forwarding Devices

In this section, we survey related work on programmable data planes in both software and hardware and compare it to our work in chapter 3.

The Click [96] modular router allows easy, rapid development of custom protocols and packet forwarding operations in software; kernel-based packet forwarding can operate at high speeds but cannot keep up with hardware for small packet sizes. An off-the-shelf NetFPGA-based router can forward traffic at 4 Gbps; this forwarding speed can be scaled by increasing the number of NetFPGA cards, and development trends suggest that much higher rates will be possible in the near future. RouteBricks [56] uses commodity processors to achieve software-based packet processing at high speed. Similarly Partridge et al. [119] use the Alpha processors to perform line card functionality.

Supercharged PlanetLab (SPP) [148] is a network processor (NP)-based technology. SPP uses Intel IXP network processors [85] for data-plane packet processing. NP-based implementations are specifically bound to the respective vendor-provided platform, which can inherently limit the flexibility of data-plane implementations.

Another solution to achieve wire-speed performance is developing custom high-speed networking chips. PLUG [39] provides a programming model for manufacturing chips to perform high-speed and flexible packet lookup. Similarly, Wolf et al. [157] propose using multiple Active Processing Units based on RISC processor cores on a single ASIC. These designs do not provide an off-the-shelf solution. Additionally, chip manufacturing is expensive: fabrication plants are not common, and cost-effective manufacturing at third-party facilities requires critical mass of demand. Thus, this development path may only make sense for large enterprises and for protocols that have already gained broad acceptance. Chip manufacturing also has a high turnaround time and post-manufacturing verification processes which can impede development of new protocols that need small development cycle and rapid deployment.

SwitchBlade is an FPGA-based platform and can be implemented on any FPGA. Its design and implementation draws inspiration from our earlier work on designing an FPGA-based data plane for virtual routers [23]. FPGA-based designs are not tied to any single vendor, and it scales as new, faster and bigger FPGAs become available. FPGAs also provide a faster development and deployment cycle compared to chip manufacturing.

Casado *et al.* argue for simple but high-speed hardware with clean interfaces to software that facilitate independent development of protocols and network hardware [42]. They argue that complex routing decisions can be made in software and cached in hardware for high-speed processing; in a sense, SwitchBlade's caching of forwarding decisions that are handled by software exception handlers embodies this philosophy. OpenFlow [117] enables the rapid development of a variety of protocols,

but the division of functions between hardware and software in SwitchBlade is quite different. Both OpenFlow and SwitchBlade provide software exceptions and caching of software decisions in hardware, but SwitchBlade also provides selectable hardware preprocessing modules which effectively moves more flexible processing to hardware. SwitchBlade also easily accommodates new hardware modules, while OpenFlow does not.

Use of Field Programmable Gate Arrays(FPGAs) to perform network functions is not new. Lockwood *et al.* [101, 147] have shown how FPGAs can be used to program modularized network functions to create data plane in FPGA. Similarly, Hadžić et al. [77] provide a platform similar to NetFPGA [10] and present a programmable packet processing architecture. Apart from differences in technologies with these works, SwitchBlade provides a virtualized packet forwarding path on FPGAs to run multiple custom protocols side by side and provides support for custom exceptions to process packets in software.

To improve FPGA programmability, NetThreads [97] programs soft microprocessors on the FPGA and then runs network packet processing code on these microprocessors. Programmability of FPGA has been an important concern there are many efforts that use different high level programming languages to program the FPGAs [35, 36, 97, 113, 131, 149]. These programmable frameworks provide support for programming FPGAs using XML [36], C [97], Click [131, 149] and custom languages [35, 113].

SwitchBlade takes a different approach to FPGA programmability. It does not provide a new programming language or programs soft microprocessors inside FPGA. Instead it uses virtualization to provide wire-speed support for parallel customized data planes, isolation between them, and their interfacing with virtualization software, which makes SwitchBlade a suitable data plane for custom data planes in existing networks. Virtualized data plane using FPGA was first presented in previous work [23]

and in [150]. SwitchBlade takes this virtualization one step ahead and shows how it can be done for multiple custom data planes in parallel with dynamic module selection. FlowVisor [139] provides some level of virtualization but sits between the OpenFlow switch and controller, essentially requiring virtualization to occur in control plane that is not scalable.

SwitchBlade assumes a model of packet forwarding devices where FPGAs are used to do fast path processing and software is used to do slow path processing. With new proposals of doing virtualized fast path processing using CPU [29, 30, 56, 58], in next section we look at work related to improving the performance of such virtualized packet forwarding devices in context of operating system(OS) virtualization [25] and compare it with our work in chapter 4.

2.1.2 Virtualized Software Packet Forwarding Devices

Many CPU scheduling algorithms in virtual machines follow process scheduling algorithms from operating systems. For example, Xen [25] has implementations of Borrowed Virtual Time (BVT) and Simple Earliest Deadline First (SEDF). Recent work [47, 76] has shown that CPU schedulers do not perform well in the presence of a combination of I/O intensive and CPU intensive applications running together on a single server. It can result in unfair scheduling of resources for CPU intensive or network I/O intensive virtual machines. Virtual machine monitors must maintain both network I/O fairness and CPU resource fairness in a virtualized environment while providing the best performance for both CPU-intensive and network-intensive applications. To enable fair sharing of CPU and network resources, we offload the task of network virtualization and fairness from the CPU and enforce it in the virtualized network interface card.

Network I/O virtualization has been used in both OS virtualization and a non-OS virtualization context. Virtual Interface Architecture (VIA) [153] is an abstract

model that is targeted towards system area networks and tries to reduce the amount of software overhead compared to traditional communication models. In traditional models, the OS kernel multiplexes the access to hardware peripherals, so all communication must involve the kernel. VIA tries to remove this communication overhead by removing the kernel operations in each communication operation. Similarly, Remote Direct Memory Access (RDMA) [129] is a data communication model that allows the network interface card direct memory access to application memory without copying data between kernel and user space.

In OS virtualization, the *driver-domain model* and the *direct I/O model* are two approaches for achieving network I/O performance while maintaining fairness. The driver-domain model [128,132] uses a separate virtual machine for device drivers. The driver domain provides better fault tolerance by isolating driver failures to a separate domain compared to maintaining device drivers in hypervisor. On the other hand, the direct I/O model gives direct hardware access to the guest domains running on the server [15, 16, 106, 130]. This approach provides near-native performance, but it sacrifices fault isolation and device transparency because it breaks the driver-domain model.

These two approaches [15, 16] provide separate queues to each virtual machine. They offload the load from the Xen software and perform multiplexing and demultiplexing in hardware. Mansley *et al.* extend the netback and netfront architecture of Xen to allow domUs to opt for *direct I/O* or "fast path" instead of driver domain or "slow path" [106]. CDNA (Concurrent, Direct Network Access) goes further by combining hardware multiplexing and demultiplexing [130]. It also assigns each virtual machine to a queue and bypasses the driver domain to provide direct network traffic I/O.

Many virtual machine monitors (e.g., Xen [25], Microsoft Hyper-V [9] and L4 [99]) use the driver-domain model to provide virtualization of I/O devices. In Xen, the

driver domain can be run separately or in domain 0. This model provides a safe execution environment for the device drivers, which enables improved fault isolation.

Both of these approaches have various problems. Although direct I/O provides better performance, it can cause reliability problems [48]. The driver-domain model is preferred because a separate domain can handle device failures without affecting the guest domains [146]. Direct I/O also requires the guest OS to have the device-specific driver, which increases the complexity of the guest OS and thus makes porting and migration more difficult. The driver-domain model, on the other hand attempts to keep the guest OS simple, but it does suffer from performance overhead, as the driver domain becomes a bottleneck, since every incoming packet has to be inside the driver domain before it can be copied to the guest domain.

In both *driver-domain model* and *direct I/O model*, interrupts are received by *hypervisor*, which dispatches them to the driver domain in case of driver domain model or to the guest domains in case of direct I/O model. In chapter 4 we preserve the driver-domain model for virtual machines and assign virtual queues to every virtual machine running on the server. Although this approach risks exposing the hypervisor to unwanted interrupts, but we maintain fairness in hardware and also suppress interrupts before sending them to the hypervisor.

In § 2.1.1 and § 2.1.2 we talked about work related to FPGA and CPU based virtualized fast path processors. In chapter 5 we talk about a system called LEGO that allows integration of these fast path processors into an existing data plane switch and section 2.1.3 discusses work related to it.

2.1.3 Virtualized Evolvable Packet Forwarding Devices

LEGO is largely complementary to the work we describe below, since it focuses on how to integrate custom packet processors/enhanced fast path processors into a single pipeline.

Recent work has developed software routers with specific packet processing capabilities based on either GPUs, FPGAs, network processors, or clusters of servers. PacketShader [79] achieves high forwarding rates for certain packet processing tasks by optimizing the packet I/O engine in Linux, creating large buffers for packets to reduce DMA overhead, processing packets in batch, and exploiting the parallelism of GPUs [53]. SwitchBlade [22] and P4 [77] make FPGA-based packet processing simpler and easier to program. But they can only support packet-processing operations that can fit onto single FPGA board, which prevents it from supporting more complex packet processing operations. Supercharged PlanetLab Platform (SPP) [148] accelerates packet forwarding by combining a general-purpose server with a network-processor subsystem. RouteBricks, PacketShader and other works [56, 79, 119] rely on multi-core CPUs to achieve fast custom packet processing. RouteBricks [56] uses the Click [96] software router to create a custom packet-processing pipeline in software. But these works despite using CPUs and GPUs don't provide a path for integration of complex functions inside the network data plane.

Software defined networking (SDN) has roots in Ethane [42], RCP [64], and 4D [74]. Although OpenFlow [108, 117] is often used synonymously with software defined networks, it is one instantiation of software defined networking with a limited set of flows and actions that is dependent on standardization process. LEGO permits a broader set of actions for processing traffic flows and a richer set of conditions for performing these actions without being dependent on standardization processes.

LEGO adds new primitives to OpenFlow's flow definitions and actions. Using specialized processing units to enhance switch functions has been proposed in other contexts, as well [75, 103, 140]. Gibb *et al.*'s OpenPipes [71] extends this notion by proposing packet processing with heterogeneous devices, but OpenPipes does not provide an abstraction for integrating heterogeneous custom packet processing and

forwarding devices; LEGO's runtime and active switching backplane, which we describe in Section 5.2, solve these problems. In contrast to previous works [102, 148] that use a single type of server peripheral card/subsystem; LEGO can incorporate heterogeneous packet processing units, since each packet processing device is merely a switch peripheral.

To manage this heterogeneity and diversity of devices, LEGO provides a runtime that can be deployed on a server to make a server's peripherals act as switch peripherals. This runtime abstracts away the diversity and handles the complexity of programming these heterogeneous devices. LEGO Runtime interfaces to a controller application, which assigns server peripherals to a switch. Managing this diversity requires supporting the "bare metal" (*e.g.*, FPGA), some instances of which may lack the resources to host large code and OS running device (*e.g.*, CPUs or NPUs). To the best of our knowledge none of the previous work handles this heterogeneity.

2.2 Top to Bottom Approach

Programmable packet forwarding devices make programmability of individual devices easy. But a network with hundreds or thousands of packet forwarding devices [137] requires a programming abstraction that can enable network operator to program the functions inside the network instead of programming functions in each packet forwarding device separately. In this dissertation, we show how our Slick programming abstraction(Chapter 6) combined with an efficient network function deployment system(Slick runtime presented in Chapter 7) can be used to program the whole network for complex network functions with minimum network resource utilization. In sections 2.2.1 and 2.2.2 we show the work related to our network function programming abstraction(Chapter 6) and its deployment system(Chapter 7).

2.2.1 Network Function Programming Abstraction

Network function virtualization (NFV) allows network operators to instantiate middleboxes in virtual machines and place those VMs at arbitrary locations in the network [61, 78]; current approaches to NFV still treat middleboxes as monolithic entities, and do not explore how the constituent components of a middlebox might be decomposed into smaller modules. Other recent work has explored how monolithic middleboxes in a cellular network might be instantiated as virtual machines [88, 156]. In contrast, Slick explores how an operator can implement individual functions in a high-level language and specifies how those functions are chained together, while remaining agnostic to how those functions are replicated and installed across the network.

Programming Model. Slick's programming model has two salient features: the decomposition of functions into modular elements and the use of triggers to redirect processing from an in-network element to the controller. Both of these features are inspired by previous work. Slick's use of the element abstraction is inspired by Click [96], which allowed programmers to write modular elements and compose them into packet processing pipelines on a single node. Slick differs from Click in that it constructs such pipelines across a network, and hence must address questions of both placement and steering. Extensible Open Middleboxes (xOMB) [20], RFC 3234 [40], and other work on modeling middleboxes [89] inspired the design and granularity of Slick element functions. Previous work has also proposed the use of triggers to allow one network element to signal to another [87, 94, 141, 144]; Slick incorporates this notion of triggers in a holistic programming model that supports more expressive triggers and perform other packet processing actions in response to the triggers. Although Slick's programming model draws inspiration from this previous work, none of these systems incorporate these mechanisms into a single coherent programming

model, as Slick does. Although OpenNF [70] and Split/Merge [127] offer programming interfaces and control-plane mechanisms for helping operators migrate existing middleboxes, they do not allow operators to write network functions that operate on specific traffic flows in the data plane, nor do they provide mechanisms for placing network functions.

Programming Languages. Many programming languages for software defined networks can be used to express network policies [66, 94, 110, 144, 154]. Most of these languages (*e.g.*, Frenetic [66], Pyretic [110], Maple [154]) provide higher-level abstractions for programming OpenFlow [108] switches. Merlin [144] and Kinetic [94] provides some abstractions for handling events that middleboxes may raise (similar to Slick's ability to process triggers), but neither provides a mechanism for installing network functions onto machines that host these functions. None of this prior work focuses on decomposing the functions provided by monolithic middleboxes into finer-grained, reusable modules, or the placement or steering functions required to implement network-wide policies with these modules.

2.2.2 Network Function Deployment

In this dissertation we divide the problem of network function deployment into following main parts *i.e.*1, *p*lacement of network functions 2, *s*teering of network traffic through these functions. Following is the survey of research work related to these areas.

Steering. Charikar *et al.* [45] and ETTM [55] assume that network functions can be placed at all machines in the network and treat resource management purely as a steering problem. This approach simplifies resource management algorithms, since placing all functions on every node reduces resource management to a traffic

steering problem. Unfortunately, as the number of middlebox functions proliferates—even Slick already supports about 15 distinct network functions—simply placing all functions on every node quickly becomes intractable.

Other recent work on steering [62, 125, 166] has assumed pre-specified, fixed placement of middleboxes within a network and focused on developing an optimal steering mechanism that minimizes utilization. In contrast, Slick makes no such assumption about placement and must thus develop mechanisms for both steering and placement. Our evaluation demonstrates that control over placement significantly reduces both path length and average link utilization. pSwitching [90] and OpenPipes [71] provide mechanisms for steering traffic through middleboxes or hardware modules but do not offer a high-level programming model and do not propose specific steering mechanisms.

Placement. Stratos explores questions of middlebox placement to reduce inter-rack traffic in data centers [68] but focuses on placement of entire virtual machines and does not explore the placement of individual network functions, as in Slick. In contrast, Slick studies a different class of placement problems that arise when middleboxes are decomposed into constituent functions, each of which may have different resource utilization and effects on traffic flows. CoMB explores whether multiple middleboxes can be consolidated on single physical machines [133] but studies consolidation at the granularity of virtual machines, as opposed to individual network functions. Our evaluation demonstrates that studying consolidation at the granularity of individual functions allows for different placement decisions (*e.g.*, placing elements that increase the amount of network traffic towards the end of a path, and vice versa), thus significantly reducing network utilization compared to CoMB. Sherry *et al.* explore placing existing network middleboxes in the cloud and routing traffic through these off-path middleboxes for processing [137]; in contrast, Slick enables on-path processing with

fine-grained network functions that an operator writes in a high-level language.

Applications. The IETF service function chaining working group is actively exploring various applications of service function chaining [135], including in mobile and data-center networks. Yang *et al.* have studied how to enable certain applications by embedding network functions in an underlying network graph, but the work focuses primarily on theoretical problems associated with embedding chains of network functions in an underlying network graph [98] and does not have a working system.

In this chapter we have talked about research work done in the community related to hardware based and software based virtualized packet forwarding fast path processors and how these cores can be integrated into evolvable packet forwarding devices. Later, in sections 2.2.1 and 2.2.2 we talked about work related to our top to bottom programming abstraction for network data planes and its runtime. In following chapters we discuss the contributions made in this dissertation for enhancing packet forwarding fast path processors and packet forwarding devices(Chapters 3, 4 and 5) and how whole network can be programmed(Chapters 6 and 7).

CHAPTER 3

VIRTUALIZED HARDWARE DATA PLANE

3.1 Motivation for Virtualized Hardware Data Plane

Countless next-generation networking protocols at various layers of the protocol stack require data-plane modifications. The past few years alone have seen proposals at multiple layers of the protocol stack for improving routing in data centers, improving availability, providing greater security, and so forth [19, 72, 111, 165]. These protocols must ultimately operate at acceptable speeds in production networks—perhaps even alongside one another—which raises the need for a platform that can support fast hardware implementations of these protocols running in parallel. This platform must provide mechanisms to deploy these new network protocols, header formats, and functions quickly, yet still forward traffic as quickly as possible. Unfortunately, the conventional hardware implementation and deployment path on custom ASICs incurs a long development cycle, and custom protocols may also consume precious space on the ASIC. Software-defined networking paradigms (e.g., Click [38, 96]) offer some hope for rapid prototyping and deployment, but a purely software-based approach cannot satisfy the strict performance requirements of most modern networks. The networking community needs a development and deployment platform that offers high performance, flexibility, and the possibility of rapid prototyping and deployment.

Although other platforms have recognized the need for fast, programmable routers, they stop somewhat short of providing a programmable platform for rapid prototyping on the hardware itself. Platforms that are based on network processors can

achieve fast forwarding performance [148], but network processor-based implementations are difficult to port across different processor architectures, and customization can be difficult if the function that a protocol requires is not native to the network processor's instruction set. All other functions should be implemented in software. PLUG [39] is an excellent framework for implementing modular lookup modules, but the model focuses on manufacturing high-speed chips, which is costly and can have a long development cycle. RouteBricks [56] provides a high-performance router, but it is implemented entirely in software, which may introduce scalability issues; additionally, prototypes developed on RouteBricks cannot be easily ported to hardware.

This chapter presents SwitchBlade, a programmable hardware platform that strikes a balance between the programmability of software and the performance of hardware, and enables rapid prototyping and deployment of new protocols. SwitchBlade enables rapid deployment of new protocols on hardware by providing modular building blocks to afford customizability and programmability that is sufficient for implementing a variety of data-plane functions. SwitchBlade's ease of programmability and wire-speed performance enables rapid prototyping of custom data-plane functions that can be directly deployed in a production network. SwitchBlade relies on field-programmable gate arrays (FPGAs). Designing and implementing SwitchBlade poses several challenges:

- *Design and implementation of a customizable hardware pipeline.* To minimize the need for resynthesizing hardware, which can be prohibitive if multiple parties are sharing it, SwitchBlade's packet-processing pipeline includes hardware modules that implement common data-plane functions. New protocols can select a subset of these modules on the fly, without resynthesizing hardware.
- *Seamless support for software exceptions.* If custom processing elements cannot be implemented in hardware (*e.g.*, due to limited resources on the hardware, such as area on the chip), SwitchBlade must be able to invoke software routines

for processing. SwitchBlade's hardware pipeline can directly invoke software exceptions on either packet or flow-based rules. The results of software processing (e.g., forwarding decisions), can be cached in hardware, making exception handling more efficient.

- *Resource isolation for simultaneous data-plane pipelines.* Multiple protocols may run in parallel on same hardware; we call each data plane a Virtual Data Plane (VDP). SwitchBlade provides each VDP with separate forwarding tables and dedicated resources. Software exceptions are the VDP that generated the exception, which makes it easier to build virtual control planes on top of SwitchBlade.
- *Hardware processing of custom, non-IP headers.* SwitchBlade provides modules to obtain appropriate fields from packet headers as input to forwarding decisions. SwitchBlade can forward packets using longest-prefix match on 32-bit header fields, an exact match on fixed length header field, or a bitmap added by custom packet preprocessing modules.

The design of SwitchBlade presents additional challenges, such as (1) dividing function between hardware and software given limited hardware resources; (2) abstracting physical ports and input/output queues; (3) achieving rate control on per-VDP basis instead of per-port basis; and (4) providing a clean interface to software.

We have implemented SwitchBlade using the NetFPGA board [10,52], but SwitchBlade can be implemented with any FPGA. To demonstrate SwitchBlade's flexibility, we use SwitchBlade to implement and evaluate several custom network protocols. We present instances of IPv4, IPv6, Path Splicing, and an OpenFlow switch, all of which can run in parallel and forward packets at line rate; each of these implementations required only modest additional development effort. SwitchBlade also provides seamless integration with software handlers implemented using Click [96], and with router slices running in OpenVZ containers [118]. Our evaluation shows that SwitchBlade

can forward traffic for custom data planes—including non-IP protocols—at hardware forwarding rates. SwitchBlade can also forward traffic for multiple distinct custom data planes in parallel, providing resource isolation for each. An implementation of SwitchBlade on the NetFPGA platform for four parallel data planes fits easily on today's NetFPGA platform; hardware trends will improve this capacity in the future. SwitchBlade can support additional VDPs with less than a linear increase in resource use, so the design will scale as FPGA capacity continues to increase.

3.2 Goals and Design Decisions

The primary goal of SwitchBlade is to enable *rapid development and deployment of new protocols working at wire-speed.* The three subgoals, in order of priority, are: (1) Enable rapid development and deployment of new protocols; (2) Provide customizability and programmability while maintaining wire-speed performance; and (3) Allow multiple data planes to operate in parallel, and facilitate sharing of hardware resources across those multiple data planes. In this section, we describe these design goals, their rationale, and highlight specific design choices that we made in SwitchBlade to achieve these goals.

Goal #1. Rapid development and deployment on fast hardware. Many next-generation networking protocols require data-plane modifications. Implementing these modifications entirely in software results in a slow data path that offers poor forwarding performance. As a result, these protocols cannot be evaluated at the data rates of production networks, nor can they be easily transferred to production network devices and systems.

Our goal is to provide a platform for designers to quickly deploy, test, and improve their designs with wire-speed performance. This goal influences our decision to implement SwitchBlade using FPGAs, which are programmable, provide acceptable speeds, and are not tied to specific vendors. An FPGA-based solution can allow

network protocol designs to take advantage of hardware trends, as larger and faster FPGAs become available. SwitchBlade relies on programmable hardware, but incorporates software exception handling for special cases; a purely software-based solution cannot provide acceptable forwarding performance. From the hardware perspective, custom ASICs incur a long development cycle, so they do not satisfy the goal of rapid deployment. Network processors offer speed, but they do not permit hardware-level customization.

Goal #2. Customizability and programmability. New protocols often require specific customizations to the data plane. Thus, SwitchBlade must provide a platform that affords enough customization to facilitate the implementation and deployment of new protocols.

Providing customizability along with fast turnaround time for hardware-based implementations is challenging: a bare-bones FPGA is customizable, but programming from scratch has a high turnaround time. To reconcile this conflict, SwitchBlade recognizes that even custom protocols share common data-plane extensions. For example, many routing protocols might use longest prefix or exact match for forwarding, and checksum verification and update, although different protocols may use these extensions on different fields in the packets. SwitchBlade provides a rich set of common extensions as modules and allows protocols to dynamically select any subset of modules that they need. SwitchBlade's modules are programmable and can operate on arbitrary offsets within packet headers.

For extensions that are not included in SwitchBlade, protocols can either add new modules in hardware or implement exception handlers in software. SwitchBlade provides hardware caching for forwarding decisions made by these exception handlers to reduce performance overhead.

Goal #3. Parallel custom data planes on a common hardware platform. The increasing need for data-plane customization for emerging network protocols makes it necessary to design a platform that can support the operation of several custom data planes that operate simultaneously and in parallel on the same hardware platform. SwitchBlade's design identifies functions that are common across data-plane protocols and provides those implementations shared access to the hardware logic that provides those common functions.

SwitchBlade allows customized data planes to run in parallel. Each data plane is called a Virtual Data Plane (VDP). SwitchBlade provides separate forwarding tables and virtualized interfaces to each VDP. SwitchBlade provides isolation among VDP using per-VDP rate control. VDPs may share modules, but to preserve hardware resources, shared modules are not replicated on the FPGA. SwitchBlade ensures that the data planes do not interface even though they share hardware modules.

Existing platforms satisfy some or all of these goals, but they do not address all the goals at once or with the prioritization we have outlined above. For example, SwitchBlade trades off higher customizability in hardware for easier and faster deployability by providing a well-defined but modular customizable pipeline. Similarly, while SwitchBlade provides parallel data planes, it still gives each data plane direct access to the hardware, and allows each VDP access to a common set of hardware modules. This level of sharing still allows protocol designers enough isolation to implement a variety of protocols and systems.

3.3 SwitchBlade Design

SwitchBlade has several unique design features that enable rapid development of customized routing protocols with wire-speed performance. SwitchBlade has a pipelined architecture (§3.3.1) with various processing stages. SwitchBlade implements Virtual Data Planes (VDP) (§3.3.2) so that multiple data plane implementations can

Table 1: SwitchBlade design features.

Feature	Design Goals	Pipeline Stages
Virtual Data Plane (§ 3.3.2)	Parallel custom data planes	VDP selection
Customizable hardware modules (§ 3.3.3)	Rapid programming, customizability	Preprocessing, Forwarding
Flexible matching in forwarding (§ 3.3.4)	Customizability	Forwarding
Programmable software exceptions (§ 3.3.5)	Rapid programming, customizability	Forwarding

be supported on the same platform with performance isolation between the different forwarding protocols. SwitchBlade provides customizable hardware modules (§3.3.3) that can be enabled or disabled to customize packet processing at runtime. SwitchBlade implements a flexible matching forwarding engine (§3.3.4) that provides a longest prefix match and an exact hash-based lookup on various fields in the packet header. There are also programmable software exceptions (§3.3.5) that can be configured from software to direct individual packets or flows to the CPU for additional processing.

3.3.1 SwitchBlade Pipeline

Figure 2 shows the SwitchBlade pipeline. There are four main stages in the pipeline. Each stage consists of one or more hardware modules. We use a pipelined architecture because it is the most straightforward choice in hardware-based architectures. Additionally, SwitchBlade is based on reference router from the NetFPGA group at Stanford [10]; this reference router has a pipelined architecture as well.

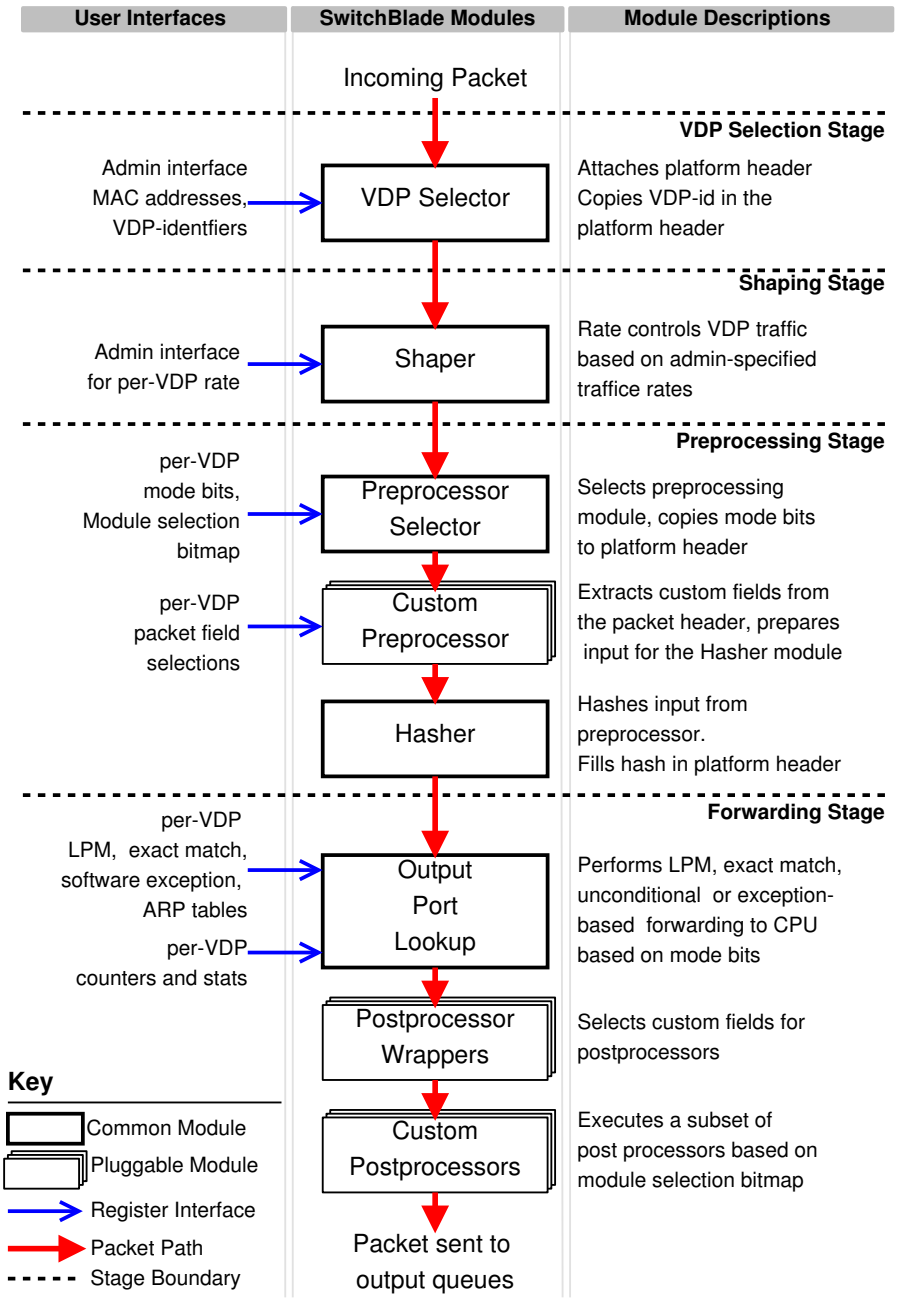


Figure 2: SwitchBlade Packet Processing Pipeline.

VDP Selection Stage. An incoming packet to SwitchBlade is associated with one of the VDPs. The *VDP Selector* module classifies the packet based on its MAC address and uses a stored table that maps MAC addresses to *VDP identifiers*. A

Table 2: Platform Header: The Mode field selects the forwarding mechanism employed by the Output Port Lookup module. The Module Selector Bitmap selects the appropriate postprocessing modules.

Field	Value	Description/Action
Mode	0	Default, Perform LPM on IPv4 destination address
	1	Perform exact matching on hash value
	2	Send packet to software for custom processing
	3	Lookup hash in software exceptions table
Module Selector Bitmap	1	Source MAC not updated
	2	Don't decrement TTL
	4	Don't Calculate Checksum
	8	Dest. MAC not updated
	16	Update IPv6 Hop Limit
	32	Use Custom Module 1
	64	Use Custom Module 2
	128	Use Custom Module 3

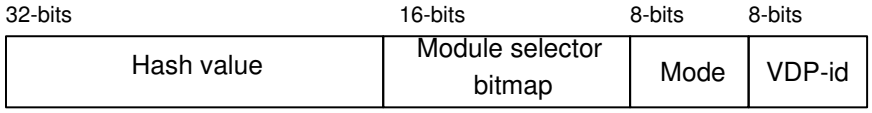


Figure 3: Platform header format. This 64 bit header is applied to every incoming packet and removed before the packet is forwarded.

register interface populates the table with the VDP identifiers and is described later. This stage also attaches a 64-bit *platform header* on the incoming packet, as shown in Figure 3. The registers corresponding to each VDP are used to fill the various fields in the platform header. SwitchBlade is a pipelined architecture, so we use a specific header format to make the architecture extensible. The first byte of this header is used to select the VDP for every incoming packet. Table 2 describes the functionality of the different fields in the platform header.

Shaping Stage. After a packet is designated to a particular VDP, the packet is sent to the shaper module. The shaper module rate limits traffic on per VDP basis. There is a register interface for the module that specifies the traffic rate limits for each VDP.

Preprocessing Stage. This stage includes all the VDP-specific preprocessing hardware modules. Each VDP can customize which preprocessor module in this stage to use for preprocessing the packet via a register interface . In addition to selecting the preprocessor, a VDP can select the various bit fields from the preprocessor using a register interface. A register interface provides information about the mode bits and the preprocessing module configurations. In addition to the custom preprocessing of the packet, this stage also has the hasher module, which can compute a hash of an arbitrary set of bits in the packet header and insert the value of the hash in the packet's platform header.

Forwarding Stage. This final stage in the pipeline handles the operations related to the actual packet forwarding. The *Output Port Lookup* module determines the destination of the packet, which could be one of: (1) longest-prefix match on the packet's destination address field to determine the output port; (2) exact matching on the hash value in the packet's platform header to determine the output port; or (3) exception-based forwarding to the CPU for further processing. This stage uses the mode bits specified in the preprocessing stage. The *Postprocessor Wrappers* and the *Custom Postprocessors* perform operations such as decrementing the packet's time-to-live field. After this stage, SwitchBlade queues the packet in the appropriate output queue for forwarding. SwitchBlade selects the postprocessing module or modules based on the *module selection bits* in the packet's platform header.

3.3.2 Custom Virtual Data Plane (VDP)

SwitchBlade enables multiple customized data planes to operate simultaneously in parallel on the same hardware. We refer to each data plane as Virtual Data Plane (VDP). SwitchBlade provides a separate packet processing pipeline, as well as separate lookup tables and register interfaces for each VDP. Each VDP may provide custom modules or share modules with other VDPs. With SwitchBlade, shared modules

are not replicated on the hardware, saving valuable resources. Software exceptions include VDP identifiers, making it easy to use separate software handlers for each VDP.

Traffic Shaping. The performance of a VDP should not be affected by the presence of other VDPs. The *shaper* module enables SwitchBlade to limit bandwidth utilization of different VDPs. When several VDPs are sharing the platform, they can send traffic through any of the four router ports. Since a VDP can start sending more traffic than what is its bandwidth limit thus affecting the performance of other VDPs. In our implementation, the shaper module comes after the *Preprocessing* stage not before it as shown in Figure 2. This implementation choice, although convenient, does not affect our results because the FPGA data plane can process packets faster than any of the inputs. Hence, the traffic shaping does not really matter. We expect, however, that in the future FPGAs there might be much more than the current four network interfaces for a single NetFPGA which would make traffic shaping of individual VDPs necessary. In the existing implementation, packets arriving at a rate greater than the allocated limit for a VDP are dropped immediately. We made this decision to save memory resources on the FPGA and to prevent any VDP from abusing resources.

Register Interface. SwitchBlade provides a register interface for a VDP to control the selection of preprocessing modules, to customize packet processing modules (*e.g.*, which fields to use for calculating hash), and to set rate limits in the shaper module. Some of the values in the registers are accessible by each VDP, while others are only available for the SwitchBlade administrator. SwitchBlade divides the register interfaces into these two security modes: the *admin* mode and the *VDP* mode. The admin mode allows setting of global policies such as traffic shaping, while the VDP mode is for per-VDP module customization.

SwitchBlade modules also provide statistics, which are recorded in the registers

and are accessible via the admin interface. The statistics are specific to each module; for example, the VDP selector module can provide statistics on packets accepted or dropped. The admin mode provides access to all registers on the SwitchBlade platform, whereas the VDP mode is only to registers related to a single VDP.

3.3.3 Customizable Hardware Modules

Rapidly deploying new routing protocols may require custom packet processing. Implementing each routing protocol from scratch can significantly increase development time. There is a significant implementation cycle for implementing hardware modules; this cycle includes design, coding, regression tests, and finally synthesis of the module on hardware. Fortunately, many basic operations are common among different forwarding mechanisms, such as extracting the destination address for lookup, checksum calculation, and TTL decrement. This commonality presents an opportunity for a design that can reuse and even allow sharing the implementations of basic operations which can significantly shorten development cycles and also save precious resources on the FPGA.

SwitchBlade achieves this reuse by providing modules that support a few basic packet processing operations that are common across many common forwarding mechanisms. Because SwitchBlade provides these modules as part of its base implementation, data plane protocols that can be composed from only the base modules can be implemented without resynthesizing the hardware and can be programmed purely using a register interface. As an example, to implement a new routing protocol such as Path Splicing [111], which requires manipulation of splicing bits (a custom field in the packet header), a VDP can provide a new module that is included at synthesis time. This module can append preprocessing headers that are later used by SwitchBlade's forwarding engine. A protocol such as OpenFlow [117] may depend only on modules that are already synthesized on the SwitchBlade platform, so it can

choose the subset of modules that it needs.

SwitchBlade's reusable modules enable new protocol developers to focus more on the protocol implementation. The developer needs to focus only on bit extraction for custom forwarding. Each pluggable module must still follow the overall timing constraints, but for development and verification purposes, the protocol developer's job is reduced to the module's implementation. Adding new modules or algorithms that offer new functionality of course requires conventional hardware development and must still strictly follow the platform's overall timing constraints.

A challenge with reusing modules is that different VDPs may need the same post-processing module (*e.g.*, decrementing TTL), but the postprocessing module may need to operate on different locations in the packet header for different protocols. In a naïve implementation, SwitchBlade would have to implement two separate modules, each looking up the corresponding bits in the packet header. This approach doubles the implementation effort and also wastes resources on the FPGA. To address this challenge, SwitchBlade allows a developer to include *wrapper modules* that can customize the behavior of existing modules, within same data word and for same length of data to be operated upon.

As shown in Figure 2 custom modules can be used in the preprocessing and forwarding stages. In the preprocessing stage, the customized modules can be selected by a VDP by specifying the appropriate selection using the register interface. Figure 4 shows an example: the incoming packet from the previous shaping stage which goes to a demultiplexer which selects the appropriate module or modules for the packet based on the input from the register interface specific to the particular VDP that the packet belongs to. After being processed by one of the protocol modules (*e.g.*, IPv6, OpenFlow), the packet arrives at the hasher module. The hasher module takes 256 bits as input and generates a 32-bit hash of the input. The hasher module need not be restricted to 256 bits of input data, but a larger input data bus would mean using

Table 3: Preprocessor Selection Codes.

Preprocessor Selection		
Code	Processor	Description
1	Custom Extractor	Allows selection of variable 64-bit fields in packet on 64-bit boundaries in first 32 bytes
2	OpenFlow	OpenFlow packet processor that allows variable field selection.
3	Path Splicing	Extracts Destination IP Address and uses bits in packet to select the Path/Forwarding Table.
4	IPv6	Extracts IPv6 destination address.

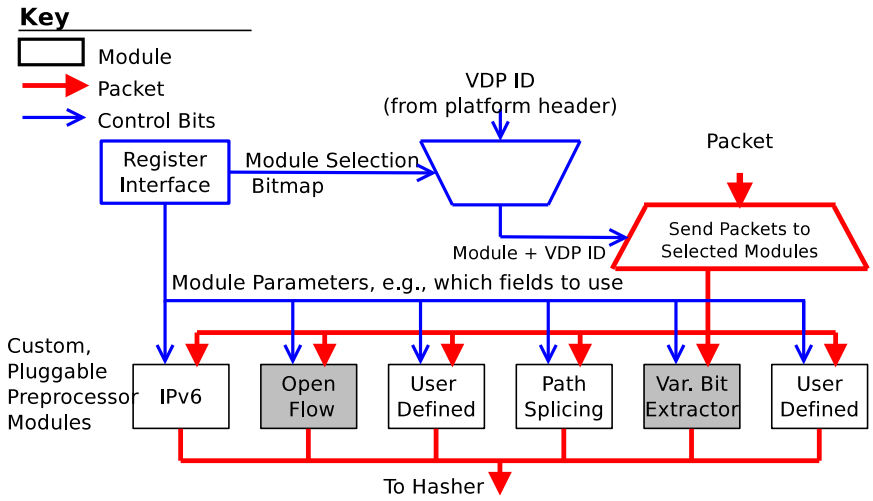


Figure 4: Virtualized, Pluggable Module for Programmable Processors.

more resources. Therefore, we decided to implement a 256-bit wide hash data bus to accommodate our design on the NetFPGA.

Each VDP can also use custom modules in the forwarding stage, by selecting the appropriate postprocessor wrappers and custom postprocessor modules as shown in Figure 2. SwitchBlade selects these modules based on the *module selection bitmap* in the platform header of the packet. Figure 5(b) shows an example of the custom wrapper and postprocessor module selection operation.

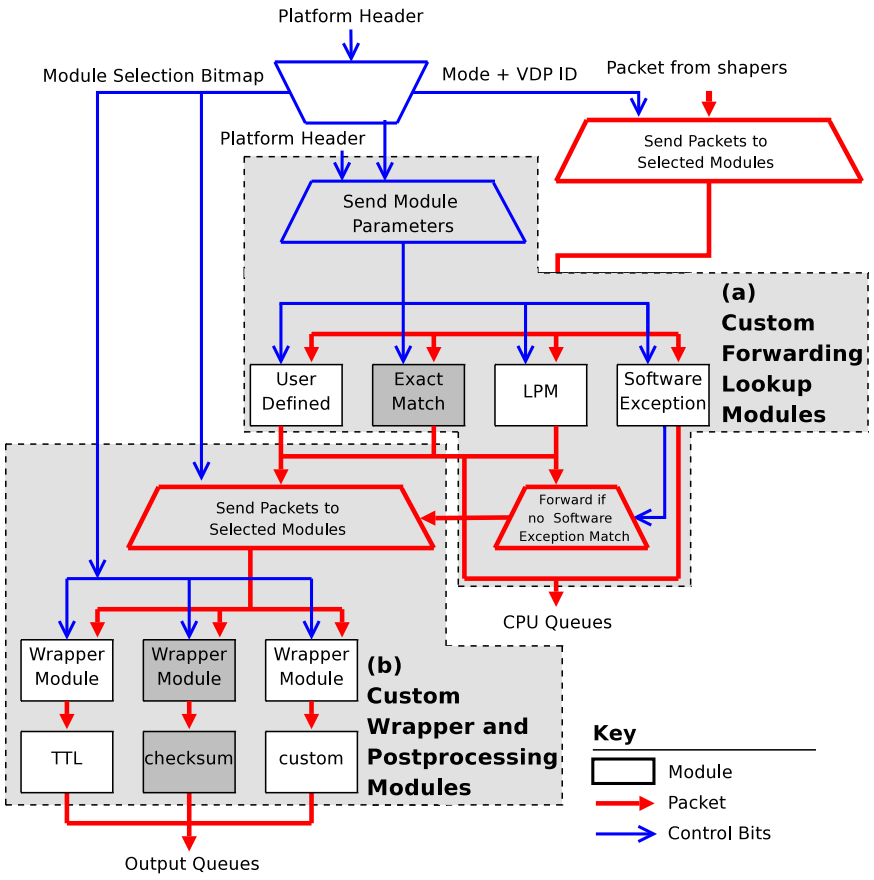


Figure 5: Output Port Lookup and Postprocessing Modules.

3.3.4 Flexible Matching for Forwarding

New routing protocols often require customized routing tables, or forwarding decisions on customized fields in the packet. For example, Path Splicing requires multiple IP-based forwarding tables, and the router chooses one of them based on splicing bits in the packet header. SEATTLE [91] and Portland [112] use MAC address-based forwarding. Some of the forwarding mechanisms are still simple enough to be implemented in hardware and can benefit from fast-path forwarding; others might be more complicated and it might be easier to just have the forwarding decision be made in software. Ideally, all forwarding should take place in hardware, but there is a tradeoff in terms of forwarding performance and hardware implementation complexity.

SwitchBlade uses a hybrid hardware-software approach to strike a balance between

forwarding performance and implementation complexity. Specifically, SwitchBlade's forwarding mechanism implementation, provided by the *Output Port Lookup* module as shown in Figure 2, provides the following four different methods for making forwarding decision on the packet: (1) conventional longest prefix matching (LPM) on any 32-bit address field in the packet header within the first 40-bytes; (2) exact matching on hash value stored in the packet's platform header; (3) unconditionally sending the packet to the CPU for making the forwarding computation; and (4) sending only packets which match certain user defined exceptions, called software exceptions(§ 3.3.5), to the CPU. The details of how the output port lookup module performs these tasks is illustrated in Figure 5(a). Modes (1) and (2) enable fast-path packet forwarding because the packet never leaves the FPGA. We observe that many common routing protocols can be implemented with these two forwarding mechanisms alone. Figure 5 is not the actual implementation but shows the functional aspect of SwitchBlade's implementation.

By default, SwitchBlade performs a longest-prefix match, assuming an IPv4 destination address is present in the packet header. To enable use of customized lookup, a VDP can set the appropriate mode bit in the *platform header* of the incoming packet. One of the four different forwarding mechanisms can be invoked for the packet by the mode bits as described in Table 2. The output port lookup module performs LPM and exact matching on the hash value from the forwarding table stored in the TCAM(Ternary Content-Addressable Memory). The same TCAM is used for LPM and for exact matching for hashing therefore the mask from the user decides the nature of match being done. Once the output port lookup module determines the output port for the packet it adds the output port number to the packet's platform header. The packet is then sent to the postprocessing modules for further processing. In Section 3.3.5, we describe the details of software work and how the packet is handled when it is sent to the CPU.

3.3.5 Flexible Software Exceptions

Although performing all processing of the packets in hardware is the only way to achieve line rate performance, it may be expensive to introduce complex forwarding implementations in the hardware. Also, if certain processing will only be performed on a few packets and the processing requirements of those packets are different from the majority of other packets, development can be faster and less expensive if those few packets are processed by the CPU instead (*e.g.*, ICMP packets in routers are typically processed in the CPU).

SwitchBlade introduces *software exceptions* to programmatically direct certain packets to the CPU for additional processing. This concept is similar to the OpenFlow concept of rules that can identify packets that match a particular traffic flow that should be passed to the controller. However, combining software exceptions with the LPM table provides greater flexibility, since a VDP can add exceptions to existing forwarding rules. Similarly, if a user starts receiving more traffic than expected from a particular software exception, that user can simply remove the software exception entry and add the forwarding rule in forwarding tables.

There is a separate exceptions table, which can be filled via a register interface on a per-VDP basis and is accessible to the output port lookup module, as shown in Figure 5(a). When the mode bits field in the platform header is set to 3 (Table 2), the output port lookup module performs an exact match of the hash value in the packet's platform header with the entries in the exceptions table for the VDP. If there is a match, then the packet is redirected to the CPU where it can be processed using software-based handlers, and if there is none then the packet is sent back to the output port lookup module to perform an LPM on the destination address. We describe the process after the packet is sent to the CPU later.

SwitchBlade's software exceptions feature allows decision caching [42]: software may install its decisions as LPM or exact match rules in the forwarding tables so that

future packets are forwarded rapidly in hardware without causing software exceptions.

SwitchBlade allows custom processing of some packets in software. There are two forwarding modes that permit this function: unconditional forwarding of all packets or forwarding of packets based on software exceptions to the CPU. Once a packet has been designated to be sent to the CPU, it is placed in a CPU queue corresponding to its VDP, as shown in Figure 5(a). The current SwitchBlade implementation forwards the packet to the CPU, with the platform header attached to the packet.

3.4 NetFPGA Implementation

In this section, we describe our NetFPGA-based implementation of SwitchBlade, as well as custom data planes that we have implemented using SwitchBlade. For each of these data planes, we present details of the custom modules, and how these modules are integrated into the SwitchBlade pipeline.

3.4.1 SwitchBlade Platform

SwitchBlade implements all the modules shown in Figure 6 on the NetFPGA [10] platform. The current implementation uses four packet preprocessor modules, as shown in Table 3. SwitchBlade uses SRAM for packet storage and BRAM and SRL16e storage for forwarding information for all the VDPs and uses the PCI interface to send or receive packets from the host machine operating system. The NetFPGA project provides reference implementations for various capabilities, such as the ability to push the Linux routing table to the hardware. Our framework extends this implementation to add other features, such as the support of virtual data planes, customizable hardware modules, and programmable software exceptions. Figure 6 shows the implementation of the NetFPGA router-based pipeline for SwitchBlade. Because our implementation is based on the NetFPGA reference implementation, adding multicast packet forwarding depends on the capabilities of NetFPGA reference router [10] implementation. Because the base implementation can support multicast forwarding,

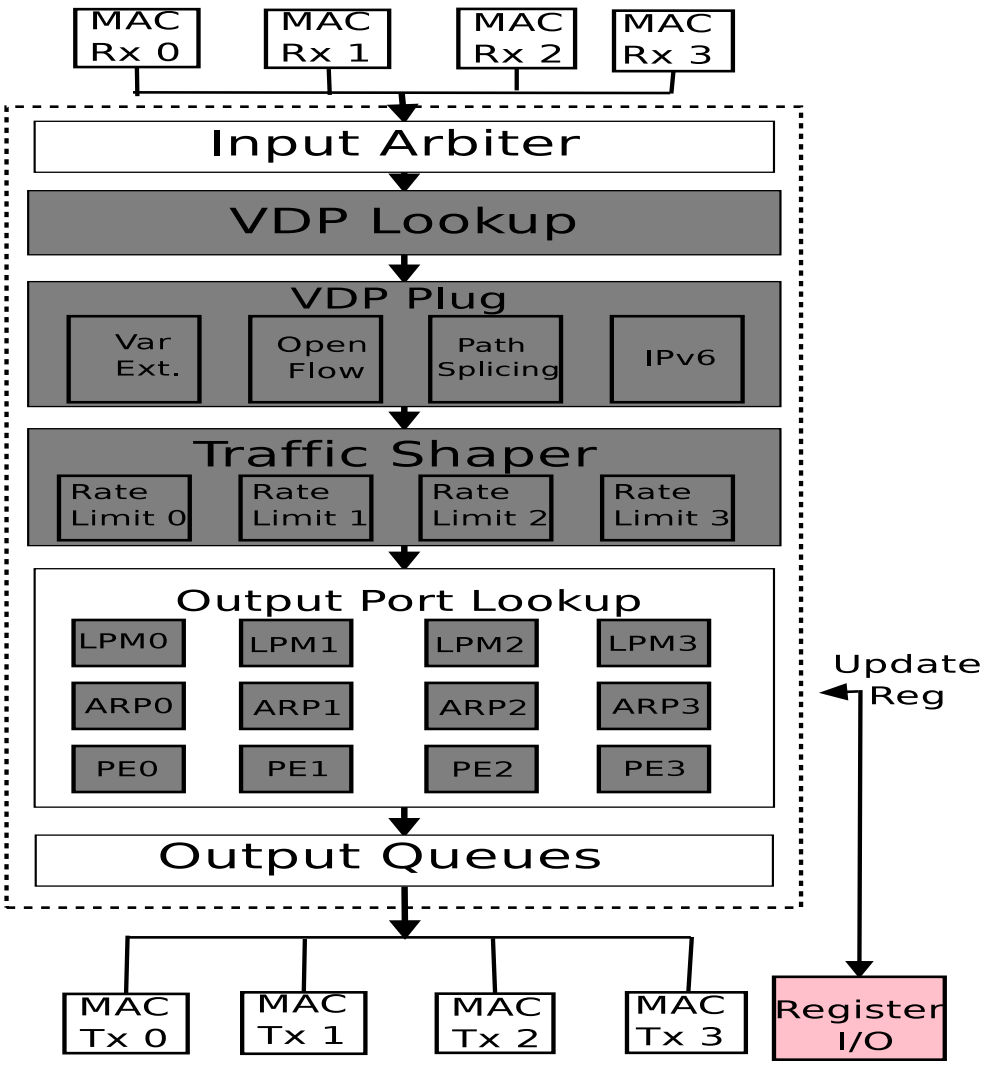


Figure 6: SwitchBlade Pipeline for NetFPGA implementation.

SwitchBlade can also support it.

VDP Selection Stage. The SwitchBlade implementation adds three new stages to the NetFPGA reference router [10] pipeline as shown in gray in Figure 6. The VDP selection stage essentially performs destination MAC lookup for each incoming packet and if the destination MAC address matches then the packet is accepted and the VDP-id is attached to the packet's platform header (Table 2). VDP selection is implemented using a CAM (Content Addressable Memory), where each MAC address is associated with a VDP-ID. This table is called the *Virtual Data Plane table*. An

admin register interface allows the SwitchBlade administrator to allow or disallow users from using a VDP by adding or removing their destination MAC entries from the table.

Preprocessing Stage. A developer can add customizable packet preprocessor modules to the VDP. There are two main benefits for these customizable preprocessor modules. First, this modularity streamlines the deployment of new forwarding schemes. Second, the hardware cost of supporting new protocols does not increase linearly with the addition of new protocol preprocessors. To enable custom packet forwarding, the preprocessing stage also provides a hashing module that takes 256-bits as input and produces a 32-bit output (Table 2). The hashing scheme does not provide a longest-prefix match; it only offers support for an exact match on the hash value. In our existing implementation each preprocessor module is fixed with one specific VDP.

Shaping Stage. We implement bandwidth isolation for each VDP using a simple network traffic rate limiter. Each VDP has a configurable rate limiter that increases or decreases the VDP's allocated bandwidth. We used a rate limiter from the NetFPGA's reference implementation for this purpose. The register interface to update the rate limits is accessible only with admin privileges.

Software Exceptions. To enable programmable software exceptions, SwitchBlade uses a 32-entry CAM within each VDP that can be configured from software using the register interface. SwitchBlade has a register interface that can be used to add a 32-bit hash representing a flow or packet. Each VDP has a set of registers to update the software exceptions table to redirect packets from hardware to software.

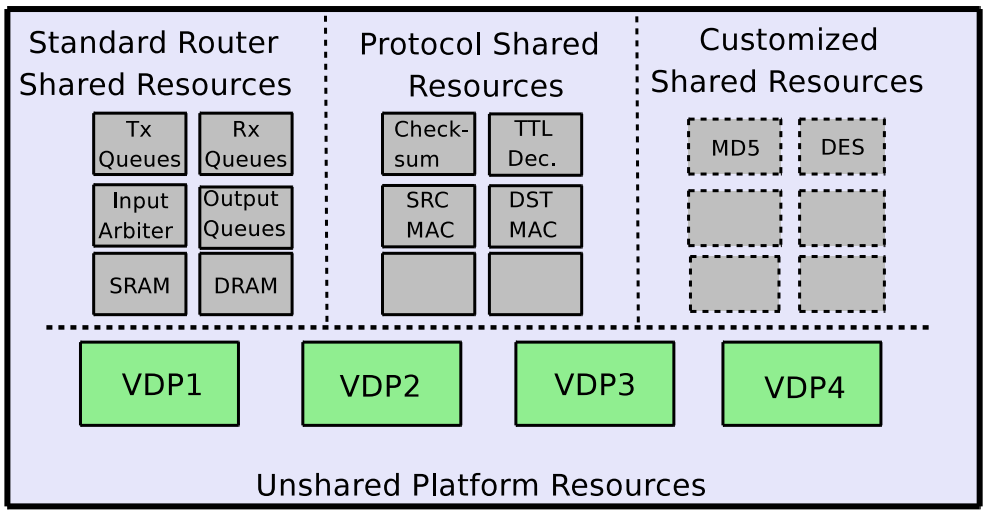


Figure 7: Resource sharing in SwitchBlade.

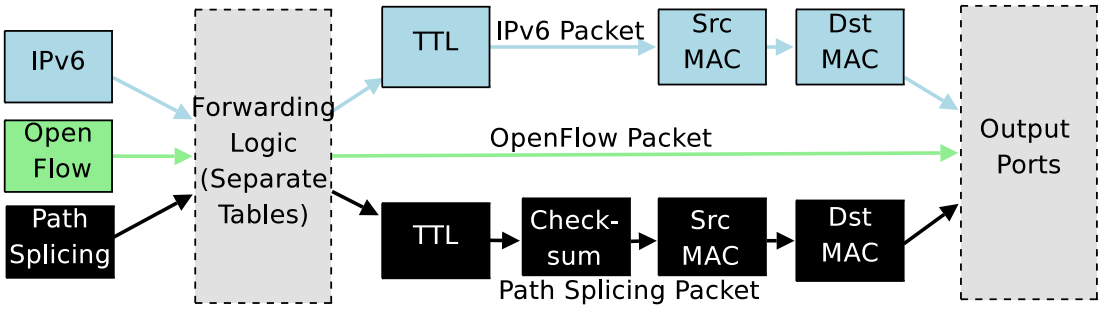


Figure 8: Life of OpenFlow, IPv6, and Path Splicing packets.

Sharing and custom packet processing. The modules that function on the virtual router instance are shared between different virtual router instances that reside on the same FPGA device. Only those modules that the virtual router user selects can operate on the packet; others do not touch the packet. This path-selection mechanism is unique. Depending on an individual virtual router user's requirements, the user can simply select the path of the packet and the modules that the virtual router user requires.

3.4.2 Custom Data Planes using SwitchBlade

Implementing any new functionality in SwitchBlade requires hardware programming in Verilog, but if the module is added as a pluggable preprocessor, then the developer

needs to be concerned with the pluggable preprocessor implementation only, as long as decoding can occur within specific clock cycles. Once a new module is added and its interface is linked with the register interface, a user can write a high-level program to use a combination of the newly added and previously added modules. Although the number of modules in a pipeline may appear limited because of smaller header size, this number can be increased by making the pipeline wider or by adding another header for every packet.

To allow developers to write their own protocols or use existing ones, SwitchBlade offers header files in C++, Perl, and Python; these files refer to register address space for that user's register interface only. A developer simply needs to include one of these header files. Once the register file is included, the developer can write a user-space program by reading and writing to the register interface. The developer can then use the register interface to enable or disable modules in the SwitchBlade pipeline. The developer can also use this interface to add hooks for software exceptions. Figure 8 shows SwitchBlade's custom packet path. We have implemented three different routing protocols and forwarding mechanisms: OpenFlow [117], Path Splicing [111], and IPv6 [54] on SwitchBlade.

OpenFlow. We implemented the exact match lookup mechanism of OpenFlow in hardware using SwitchBlade without VLAN support. The OpenFlow preprocessor module, as shown in Figure 4, parses a packet and extracts the ten tuples of a packet defined in OpenFlow specifications. The OpenFlow preprocessor module extracts the bits from the packet header and returns a 240-bit wide OpenFlow flow entry. These 240-bits travel on a 256-bit wire to the hasher module. The hasher module returns a 32-bit hash value that is added to the SwitchBlade platform header (Figure 3). After the addition of hash value this module adds a module selector bitmap to the packet's platform header. The pipeline then sets mode field in the packet's platform header

to 1, which makes the output port lookup module perform an exact match on the hash value of the packet. The output port lookup module looks up the hash value in the exact match table and forwards the packet to the output port if the lookup was a hit. If the table does not contain the corresponding entry, the platform forward the packet to the CPU for processing with software-based handlers.

Because OpenFlow offers switch functionality and does not require any extra post-processing (*e.g.*, TTL decrement or checksum calculation), a user can prevent the forwarding stage from performing any extra postprocessing functions on the packet. Nothing happens in the forwarding stage apart from the lookup, and SwitchBlade queues the packet in the appropriate output queue. A developer can update source and destination MACs as well, using the register interface.

Path Splicing. Path Splicing enables users to select different paths to a destination based on the *splicing bits* in the packet header. The splicing bits are included as a bitmap in the packet's header and serve as an index for one of the possible paths to the destination. To implement Path Splicing in hardware, we implemented a processing module in the preprocessing stage. For each incoming packet, the preprocessor module extracts the splicing bits and the destination IP address. It concatenates the IP destination address and the splicing bits to generate a new address that represents a separate path. Since Path Splicing allows variation in path selection, this bit field can vary in length. The hasher module takes this bit field, creates a 32-bit hash value, and attaches it to the packet's platform header.

When the packet reaches the exact match lookup table, its 32-bit hash value is extracted from SwitchBlade header and is looked up in the exact match table. If a match exists, the card forwards the packet on the appropriate output port. Because the module is concatenating the bits and then hashing them and there is an exact match down the pipeline, two packets with the same destination address but different

paths will have different hashes, so they will be matched against different forwarding table entries and routed along two different paths. Since Path Splicing uses IPv4 for packet processing, all the postprocessing modules on the default path (*e.g.*, TTL decrement) operate on the packet and update the packet's required fields. SwitchBlade can also support equal-cost multipath (ECMP). For this protocol, the user must implement a new preprocessor module that can select two different paths based on the packet header fields and can store their hashes in the lookup table sending packets to two separate paths based on the hash match in lookup.

IPv6. The IPv6 implementation on SwitchBlade also uses the customizable preprocessor modules to extract the 128-bit destination address from an incoming IPv6 packet. The preprocessor module extracts the 128-bits and sends them to the hasher module to generate a 32-bit hash from the address.

Our implementation restricts longest prefix match to 32-bit address fields, so it is not currently possible to perform longest prefix match for IPv6. The output port lookup stage performs an exact match on the hash value of the IPv6 packet and sends it for postprocessing. When the packet reaches the postprocessing stage, it only needs to have its TTL decremented because there is no checksum in IPv6. But it also requires to have its source and destination MACs updated before forwarding. The module selector bitmap shown in figure 3 and table 2 enables only the postprocessing module responsible for TTL decrement and not the ones doing checksum recalculation. Because the TTL offset for IPv6 is at a different byte offset than the default IPv4 TTL field, SwitchBlade uses a wrapper module that extracts only the bits of the packet's header that are required by the TTL decrement module; it then updates the packet's header with the decremented TTL.

3.5 Evaluation

In this section, we evaluate our implementation of SwitchBlade using NetFPGA [10] as a prototype development platform. Our evaluation focuses on three main aspects of SwitchBlade: (1) resource utilization for the SwitchBlade platform; (2) forwarding performance and isolation for parallel data planes; and (3) data-plane update rates.

3.5.1 Resource Utilization

To provide insight about the resource usage when different data planes are implemented on SwitchBlade, we used Xilinx ISE [161] 9.2 to synthesize SwitchBlade. We found that a single physical IPv4 router implementation developed by the NetFPGA group at Stanford University uses a total of $23K$ four-input LUTs, which consume about 49% of the total available four-input LUTs, on the NetFPGA. The implementation also requires 123 BRAM units, which is 53% of the total available BRAM.

We refer to our existing implementation with one OpenFlow, one IPv6, one variable bit extractor, and one Path Splicing preprocessor with an IPv4 router and capable of supporting four VDPs as the SwitchBlade"base configuration". This implementation uses $37K$ four-input LUTs, which account for approximately 79% of four-input LUTs. Approximately 4.5% of LUTs are used for shift registers. Table 4 shows the resource utilization for the base SwitchBlade implementation; SwitchBlade uses more resources than the base IPv4 router, as shown in table 5, but the increase in resource utilization is less than linear in the number of VDPs that SwitchBlade can support.

Sharing modules enables resource savings for different protocol implementations. Table 5 shows the resource usage for implementations of an IPv4 router, an OpenFlow switch, and path splicing. These implementations achieve 4 Gbps; OpenFlow and Path Splicing implementations provide more resources than SwitchBlade. But there is not much difference in resource usage for these implementations when compared with the possible configurations which SwitchBlade can support.

Table 4: Resource utilization for the base SwitchBlade platform.

Resource	NetFPGA Utilization	% Utilization
Slices	21 K out of 23 K	90%
4-input LUTs	37 K out of 47 K	79%
Flip Flops	20 K out of 47 K	42%
External IOBs	353 out of 692	51%
Eq. Gate Count	13 M	N/A

Table 5: Resource usage for different data planes.

NetFPGA Implementation	Slices	4-input LUTs	Flip Flops	BRAM	Equivalent Gate Count
Path Splicing	17 K	19 K	17 K	172	12 M
OpenFlow	21 K	35 K	22 K	169	12 M
IPv4	16 K	23 K	15 K	123	8 M

Virtual Data Planes can support multiple forwarding planes in parallel. Placing four Path Splicing implementations in parallel on a larger FPGA to run four Path Splicing data planes will require four times the resources of existing Path Splicing implementation. Because no modules are shared between the four forwarding planes, the number of resources will not increase linearly and will remain constant in the best case.

From a gate count perspective, Path Splicing with larger forwarding tables and more memory will require approximately four times the resources as in Table 5; SwitchBlade with smaller forwarding tables and less memory will require almost same amount of resources. This resource usage gap begins to increase as we increase the number of Virtual Data Planes on the FPGA. Recent trends in FPGA development such as Xilinx Virtex 7 [158] and Xilinx Virtex UltraScale [160] suggest higher speeds and larger area; these trends will allow more VDPs to be placed on a single FPGA, which will facilitate more resource sharing.

3.5.2 Forwarding Performance and Isolation

We used the NetFPGA-based packet generator [51] for traffic generation to generate high speed traffic to evaluate the forwarding performance of SwitchBlade and the

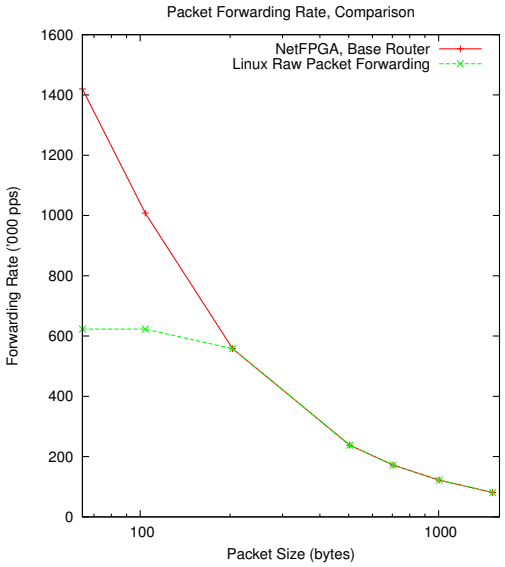


Figure 9: Packet forwarding rates (NetFPGA Hardware Router vs Linux Raw Packet Forwarding).

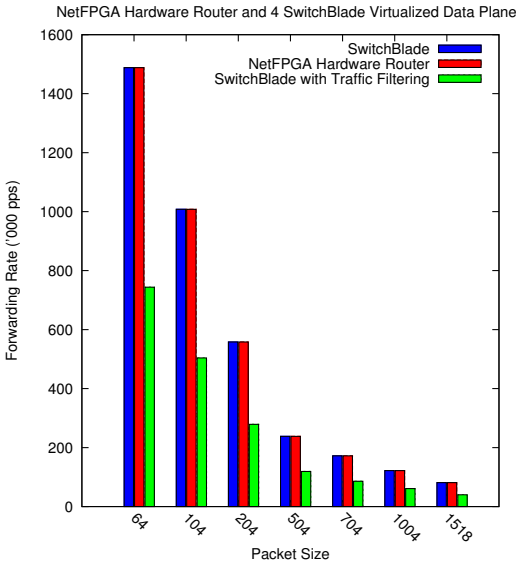


Figure 10: Data plane performance: NetFPGA reference router vs. SwitchBlade.

isolation provided between the VDPs. Some of the results we present in this section are derived from experiments in previous work [23].

Raw forwarding rates. Previous work has measured the maximum sustainable packet forwarding rate for different configurations of software-based virtual routers [29]. We also measure packet forwarding rates and show that hardware-accelerated forwarding can increase packet forwarding rates. We compare forwarding rates of Linux and NetFPGA-based router implementation from NetFPGA group [10], as shown in Figure 9. The maximum forwarding rate shown, about 1.4 million packets per second, is the maximum traffic rate which we were able to generate through the NetFPGA-based packet generator.

The Linux kernel drops packets at high loads, but our configuration could not send packets at a high enough rate to see packet drops in hardware. If we impose the condition that no packets should be dropped at the router, then the packet forwarding rates for the Linux router drops significantly, but the forwarding rates for

Table 6: Physical Router, Packet Drop Behavior.

Physical Router ('000s of packets)			
Packet Size(bytes)	Pkts Sent	Pkts Fwd @ Core	Pkts recv @Sink
64	40 K	40 K	20 K
104	40 K	40 K	20 K
204	40 K	40 K	20 K
504	40 K	40 K	20 K
704	40 K	40 K	20 K
1004	39.8 K	39.8 K	19.9 K
1518	4 K	4 K	1.9 K

the hardware-based router remain constant. Figure 9 shows packet forwarding rates when this "no packet drop" condition is *not* imposed (*i.e.*, we measure the maximum sustainable forwarding rates). For large packet sizes, SwitchBlade could achieve the same forwarding rate using in-kernel forwarding as we were using a single port of NetFPGA router. Once the packet size drops below 200 bytes; the software-based router cannot keep pace with the forwarding requirements.

Forwarding performance for Virtual Data Planes. Figure 10 shows the data-plane forwarding performance of SwitchBlade running four data planes in parallel versus the NetFPGA reference router [10], for various packet sizes. We have disabled the rate limiters in SwitchBlade for these experiments. The figure shows that running SwitchBlade incurs no additional performance penalty when compared to the performance of running the reference router [10]. By default, traffic belonging to any VDP can arrive on any of the physical Ethernet interfaces since all of the ports are in promiscuous mode. To measure SwitchBlade's to filter traffic that is not destined for any VDP, we flooded SwitchBlade with a mix of traffic where half of the packets had destination MAC addresses of SwitchBlade virtual interfaces and half of the packets had destination MAC addresses that didn't belong to any VDP. As a result, half of the packets were dropped and rest were forwarded, which resulted in a forwarding rate that was half of the incoming traffic rate.

Table 7: Four Parallel Data Planes, Packet Drop Behavior.

Four Data Planes ('000s of packets)			
Packet Size(bytes)	Pkts Sent	Pkts Fwd @ Core	Pkts recv @Sink
64	40 K	40 K	20 K
104	40 K	40 K	20 K
204	40 K	40 K	20 K
504	40 K	40 K	20 K
704	40 K	40 K	20 K
1004	39.8 K	39.8 K	19.9 K
1518	9.6 K	9.6 K	4.8 K

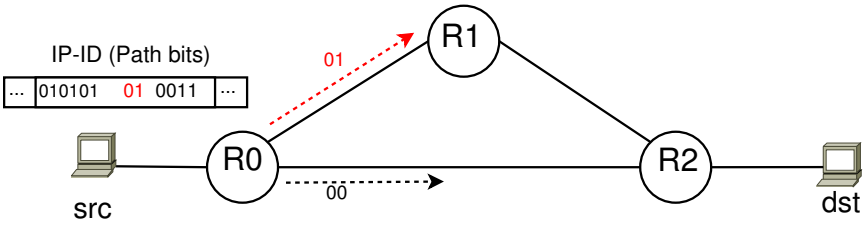


Figure 11: Test topology for testing SwitchBlade implementation of Path Splicing.

Isolation for Virtual Data Planes. To measure CPU isolation, we used four parallel data planes to forward traffic when a user-space process used 100% of the CPU. We then sent traffic where each user had an assigned traffic quota in packets per second. When no user surpassed the assigned quotas, the router forwarded traffic according to the assigned rates, with no packet loss. To measure traffic isolation, we set up a topology where two 1 Gbps ports of routers were flooded at 1 Gbps and a sink node were connected to a third 1 Gbps port. We used four VDPs to forward traffic to the same output port. Tables 6 and 7 show that, at this packet forwarding rate, only half of the packets make it through, on first come first serve basis, as shown in fourth column. These tables show that both the reference router implementation and SwitchBlade have the same performance in the worst-case scenario when an output port is flooded. The second and third columns show the number of packets sent to the router and the number of packets forwarded by the router. Our design does not prevent against contention that may arise when many users send traffic to one output port.

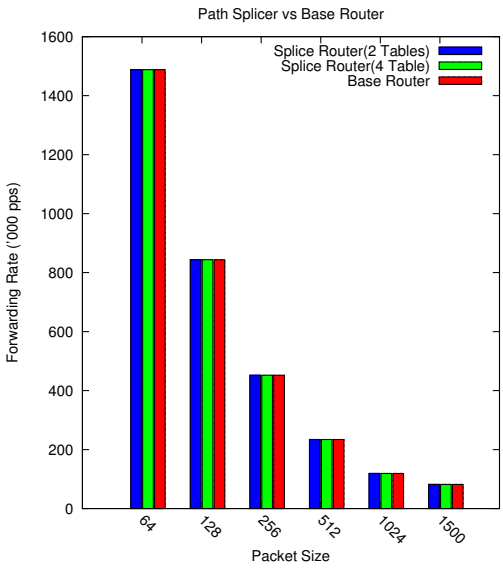


Figure 12: Path Splicing router performance with varying load compared with base router.

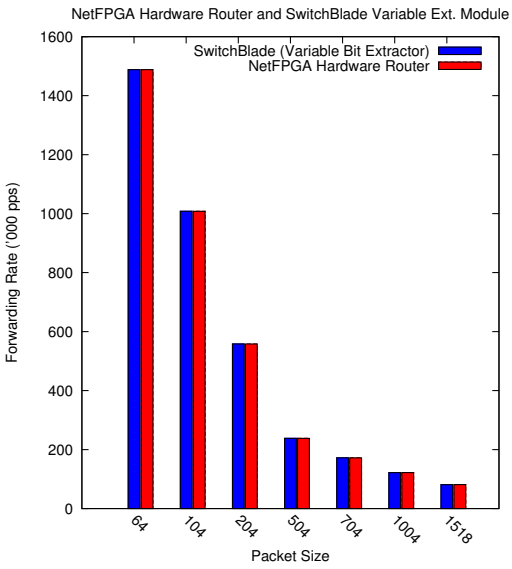


Figure 13: Variable Bit Length Extraction router performance compared with base router.

Forwarding performance for non-IP packets. We also tested whether SwitchBlade incurred any forwarding penalty for forwarding custom, non-IP packets; SwitchBlade was also able to forward these packets at the same rate as regular IP packets. Figure 11 shows the testbed we used to test the Path Splicing implementation. We again used the NetFPGA-based hardware packet generator [51] to send and receive traffic. Figure 12 shows the packet forwarding rates of this NetFPGA-based implementation, as observed at the sink node. No packet loss occurred on any of the nodes shown in Figure 11. We sent two flows with same destination IP address but using different splicing bits to direct them to different routers. Packets from one flow were sent to $R2$ via $R1$, while others went directly to $R2$. In another iteration, we introduced four different flows in the network, such that all four forwarding tables at router $R0$ and $R2$ were looked up with equal probability; in this case, SwitchBlade also forwarded the packets at full rate. Both these experiments show that SwitchBlade can implement schemes like Path Splicing and forward traffic at hardware speeds for non-IP packets.

Table 8: Forwarding Table Update Performance.

VDPs	Total Ent.	Entries/Table	Time(sec)	Single Ent. (μs)
1	400 K	400 K	86.582	216
2	800 K	400 K	86.932	112
3	1,200 K	400 K	88.523	74
4	1,600 K	400 K	89.770	56

In another experiment, we used the Variable Bit Extraction module to extract first 64 bits from the header for hashing. We used a simple source and sink topology with SwitchBlade between them and measured the number of packets forwarded. Figure 13 shows the comparison of forwarding rates when forwarding was being done using SwitchBlade based on the first 64-bits of an Ethernet frame and when it was done using NetFPGA base router.

3.5.3 Data-Plane Update Rates

Each VDP in a router on SwitchBlade needs to have its own forwarding table. Because the VDPs share a single physical device, simultaneous table updates from different VDPs might create a bottleneck. To evaluate the performance of SwitchBlade for forwarding table update speeds, we assumed the worst-case scenario, where all VDPs flush their tables and rewrite them again at the same time. We assumed that the table size for each VDP is 400,000 entries. We updated all four tables simultaneously, but there was no performance decrease while updating the forwarding table from software. Four processes were writing the table entries in the forwarding table.

Table 8 shows updating 1.6 million entries simultaneously took 89.77 seconds on average, with a standard deviation of less than one second. As the number of VDPs increases, the average update rate remains constant, but as the number of VDPs increases, the PCI interconnect speed becomes a bottleneck between the VDP processes updating the table and the SwitchBlade FPGA.

3.6 Summary

We have presented the design, implementation, and evaluation of SwitchBlade, a platform for deploying custom protocols on programmable hardware. SwitchBlade uses a pipeline-based hardware design; using this pipeline, developers can swap common hardware processing modules in and out of the packet-processing flow on the fly, without having to resynthesize hardware. SwitchBlade also offers programmable software exception handling to allow developers to integrate custom functions into the packet processing pipeline that cannot be handled in hardware. SwitchBlade's customizable forwarding engine also permits the platform to make packet forwarding decisions on various fields in the packet header, enabling custom, non-IP based forwarding at hardware speeds. Finally, SwitchBlade can host multiple data planes in hardware in parallel, sharing common hardware processing modules while providing performance isolation between the respective data planes. These features make SwitchBlade a suitable platform for hosting virtual routers or for simply deploying multiple data planes for protocols or services that offer complementary functions in a production environment. We implemented SwitchBlade using the NetFPGA platform, but SwitchBlade can be implemented with any FPGA.

CHAPTER 4

VIRTUALIZED SOFTWARE DATA PLANE

4.1 Introduction

Recent work has shown how virtual machines [58,59,107] can be used to have multiple parallel forwarding data planes in software. An important aspect of virtual machine design and implementation is *fairness* in resource allocation across virtual machines. Although it is relatively well understood how to fairly allocate computing resources like CPU cycles, the notion of fairness becomes less clear when it is applied to I/O—and, in particular, network I/O. A common approach for controlling I/O resource utilization is to implement scheduling in the virtual machine monitor or hypervisor. Unfortunately, this approach induces significant CPU overhead due to I/O interrupt processing.

Various approaches to increase the performance of virtual-machine I/O have been proposed. Some try to provide new scheduling algorithms for virtual machines for better network I/O performance. Other techniques use existing schedulers and try to provide system optimizations, both in software and in hardware. From the operating system perspective, they fall into two categories: the driver-domain model and those that provide direct I/O access for high speed network I/O.

A key problem in virtual machines is *network I/O fairness*, which guarantees that no virtual machine can have disproportionate use of the physical network interface. This chapter presents a design that achieves network I/O fairness across virtual machines by applying rate limiting in hardware on virtual interfaces. We show that applying rate limiting in hardware can reduce the CPU cycles required to implement

per-virtual machine rate limiting for network I/O. Our design applies generally to network I/O fairness for network interfaces in general, making our contributions applicable to any setting where virtual machines need network I/O fairness (i.e., either for virtual servers [25] or virtual routers [58]).

Our goals for the implementation of network I/O fairness for virtual machines are four-fold. First, the rate limiter should be *fair*: it should be able to enforce network I/O fairness across different virtual machines. Second, the mechanism must be *scalable*: the mechanism must scale as the number of virtual machines increases. Third, it must be *flexible*: because virtual machines may be continually remapped to the underlying hardware, the mechanism must operate in circumstances where virtual ports may be frequently remapped to underlying physical ports; this association should also be easily *programmable*. Finally, the mechanism must be *robust*, so that neither benign users nor malicious attackers can subvert the rate-control mechanism.

We achieve these design goals with a simple principle: *push rate limiting as close as possible to the underlying physical hardware, suppressing as many software interrupts as possible*. Our implementation builds on our previous work in fast data planes for virtual routers that are built on commodity hardware [23]. The design suppresses interrupts by implementing rate limiting for each virtual queue in hardware, preventing the hypervisor or operating system from ever needing to process the packets or implement any fairness mechanisms. Our evaluation shows that implementing rate limiting directly in hardware, rather than relying on the virtual machine hypervisor, can reduce both CPU interrupts and the required number of instructions to forward packets at a certain rate by an order of magnitude.

The rest of this chapter proceeds as follows. Section 4.3 presents the basic design of a virtualized network interface card to enable network resource isolation in hardware and freeing CPU from unwanted packets overhead to do better virtual machine resource scheduling; this design is agnostic to any specific programmable hardware

platform. Section 4.4 presents an implementation of our design using the NetFPGA platform. Section 4.5 presents a preliminary evaluation of the virtualized network interface card; Section 4.6 concludes this chapter with summary of contributions and lessons learned.

4.2 Design Goals

This section outlines the design goals and challenges for the virtualized network interface card that provides both network I/O fairness and can support multiple virtual machines running on the CPU of data plane device. These VMs can act as virtualized data planes or can act as end host machines.

1. **Fair.** Our main goal is to provide network I/O fairness to all users who share a single physical server. The design should be scalable enough to provide both transmit and receive-side fairness to all users on a single server. Fair access to the network resources enables a cloud service provider to allocate a fixed amount of bandwidth to each user at the server level, so that as load increases, no user should have access to an unfair share of resources.
2. **Scalable.** Increasing CPU processing power [14] with increasing interconnection bandwidths (e.g, PCIe 2.0) means that data can be sent to the CPU at increasingly high rates. These two facts point towards a higher per-server network bandwidth in cloud infrastructure and data-center networks. A general rule of thumb of one virtual machine per thread per processor [112] means that a single server might host tens of virtual machines. Therefore, the design should be scalable enough to handle the bandwidth of an increasing number of virtual machines per server.
3. **Flexible.** The design should allow the physical server owner to dynamically change the physical port association of a virtual Ethernet card on the fly. A physical card might have an unequal number of physical and virtual Ethernet

ports, which means that number of virtual Ethernet cards that can be maintained on a single card should not depend on physical Ethernet ports on the hardware. We assume that new virtual machines will be created regularly on a server and that they can have new MAC addresses with each new instance. Similarly, if a virtual machine migrates from one server to another, it might want to keep its MAC address as it moves; therefore, the network interface card should be able to accommodate the new MAC address if needed.

4. **Programmable.** Associating each virtual Ethernet interface should be based on MAC addresses. The virtual network interface card should be programmable enough to allow the administrator to associate the physical and virtual interfaces using a simple programmable interface. It should also enable the administrator to associate virtual Ethernet interfaces with physical ones based on user defined values (e.g, IPv4 addresses).

5. **Robust.** Design should be robust enough to not introduce any malicious behavior. e.g., Separating physical interfaces from virtual Ethernet interfaces opens the possibility of Denial of Service (DoS) attacks on the VMs on the same physical server. We aim for our design to be robust enough to withstand any malicious activity from the users residing on the same physical server.

The next sections describe the design and implementation that tries to achieve these goals.

4.3 Design

Our design assumes multiple users on a server sharing a physical network interface card, as shown in Figure 14. The higher per-server network bandwidth requirement of cloud infrastructure and data-center networks makes scalability one of the main issues for our design. We use a CAM-based design to handle multiplexing and demultiplexing of high-speed network traffic in hardware for a large number of virtual

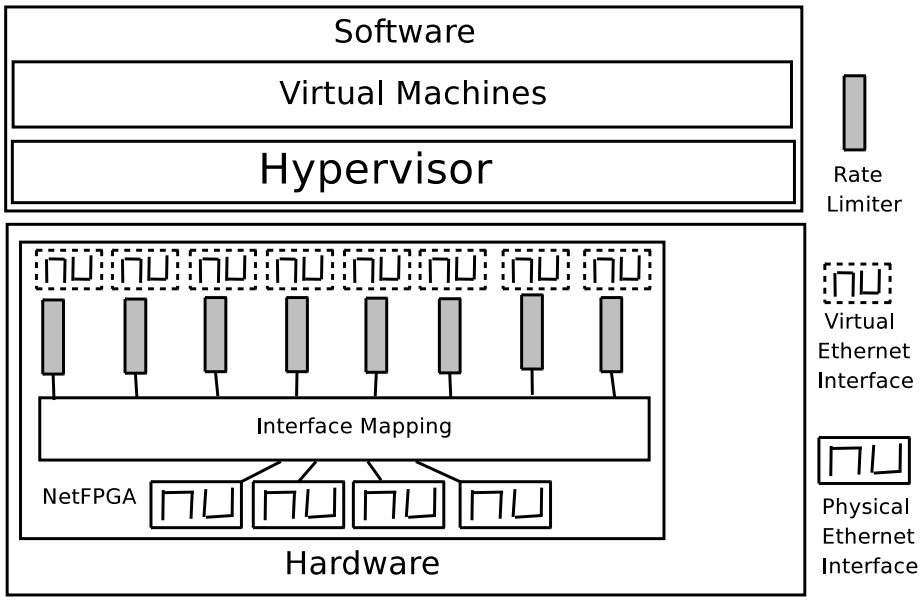


Figure 14: Virtualized network interface card with rate limiters.

machines.

One of our main goals is to maintain network I/O fairness in hardware and suppress unwanted interrupts on the card before sending packets to the CPU. Achieving this network I/O fairness in hardware can ultimately result in better performance and better resource scheduling for the server. Virtualized Ethernet cards available in the market [15, 16] provide a notion of virtual Ethernet interfaces, as we discuss in Section 4.3.1. We also explain how virtual interfaces can be mapped to the physical interfaces on a card (Section 4.3.2). Previous work has shown that multi-queue network interface cards provide better performance than single-queue cards [56, 128]. Because there is no open implementation of a virtualized network interface card, we first modify the NetFPGA [10] NIC implementation to provide a virtualized network interface card. We further modify this implementation to provide programmability and network I/O fairness, suppressing the interrupts inside hardware before sending them to the CPU (Section 4.3.3).

4.3.1 Virtualized Ethernet Card

Virtualized network interface cards (e.g., [15, 16]) provide packet multiplexing and demultiplexing in hardware instead of software. Without hardware support, this multiplexing and demultiplexing occurs in the driver domain; with increasing network bandwidth requirements for server, multiplexing and demultiplexing in software can easily become a performance bottleneck. Because physical Ethernet cards can have many virtual network interfaces, we implement the packet multiplexing and demultiplexing using the MAC mapping table and map each virtual network interface in VM to a physical queue in hardware.

In the NetFPGA reference design for network interface card, each Ethernet port is mapped to a corresponding Ethernet interface in software. Thus, if a packet arrives on a physical Ethernet interface, by default it is sent to the software interface in Linux kernel, at which point the user process handles the packet.

We have allocated virtual Ethernet interfaces to physical queues using a mapping table, as shown in Figure 14. Because the number of virtual Ethernet interfaces in software can be more than the physical ports available on hardware we use a table that maps physical to virtual Ethernet interface mappings so that the packet can be sent to the appropriate queue in hardware.

Each Ethernet interface must also have its own MAC address through which it can be identified to the outside world. Thus, each virtual Ethernet interface has a 48-bit register that stores the MAC address for the virtual Ethernet interface.

4.3.2 Mapping Ethernet Interfaces

On a physical card, the number of physical Ethernet interfaces may not be equal to virtual Ethernet interfaces. Therefore, to identify each virtual Ethernet interface uniquely, each virtual interface must have a unique MAC address. MAC addresses for the virtual Ethernet interfaces are maintained in a small table. For every incoming

packet, its destination MAC address is looked up in the table to see if the packet belongs to one of the virtual Ethernet interfaces on the card. If there is a hit in this table, it means the packet is addressed to one of the virtual machines and is accepted; in case of a miss, the network interface card drops the packet.

In addition to book-keeping of MAC addresses for virtual interfaces, this table also maps each MAC address to the corresponding virtual Ethernet interface. This mapping translates the MAC address to physical queue mapping in hardware. For each incoming packet, if its destination MAC address is present in the mapping table, a table lookup returns the corresponding virtual Ethernet interface to the MAC address. Based on this value, the incoming packet is sent to the appropriate virtual interface in software.

The mapping of physical to Virtual Ethernet interfaces is dynamic and can be changed by the administrator on the fly. In addition to redirecting received packets, the mapping table has information about outgoing packets. It has a field that is looked up for every outgoing packet. This field makes sure that users can send traffic out of the physical interface through which they are allowed to send traffic out instead of any other interface.

4.3.3 Fairness and Interrupt Suppression

Each virtual Ethernet interface's receive queue has rate limiters placed on it. Once the packet is forwarded from the mapping tables stage, it reaches to a token bucket rate limiter as shown in figure 14. These rate limiters provide a soft limit to the traffic coming to the CPU and these limits can be changed by the administrator using a register interface through a user-space program.

Rate limiting before the receive queue of each virtual Ethernet interface can help enforce a fairness policy set by the administrator. If any of the user tries to send more traffic than what is allocated than the packets are dropped. This design ensures

that different virtual machine users receive no more than their allocated share of bandwidth.

In addition to providing network traffic fairness between the virtual machines, packet dropping in hardware makes sure that no extra packets go the hypervisor. There are two reason to drop packets in hardware. First of all bandwidth of NIC interconnect interface may not be enough to send all traffic from virtualized NIC to the software e.g. PCI bandwidth is much less than 4Gbps bandwidth of NetFPGA card. Therefore NIC should not be receiving more traffic than it can push through the interconnect interface. Rate limiter imposes this limit and drops any extra packets that can not be pushed through the interconnect. Secondly, it puts a limit on bandwidth usage by each VM thus there are no extra interrupts generated for the hypervisor to handle extra traffic for a specific user.

Inter-virtual machine traffic on the same server can both be legitimate and illegitimate. Any legitimate virtual machine user on a server can mount malicious attacks on the neighbors residing on the same physical server. Its possible that a user can send a packet with his source MAC address, destined to another machine on the same server. This kind of attack will result in wasted CPU cycles for the user that is under attack. One solution can be to simply block inter-virtual machine traffic, but another solution can be to block inter virtual machine traffic based on legitimacy of inter-virtual machine traffic. A simple blocking solution mimics the behavior in servers with different NICs and can implemented by looking up the source and destination MAC addresses for each outgoing packet. If the destination MAC address and the source MAC address of packet are matched then the packet can be dropped; if there is a match for only one of the addresses, then the packet can be transmitted.

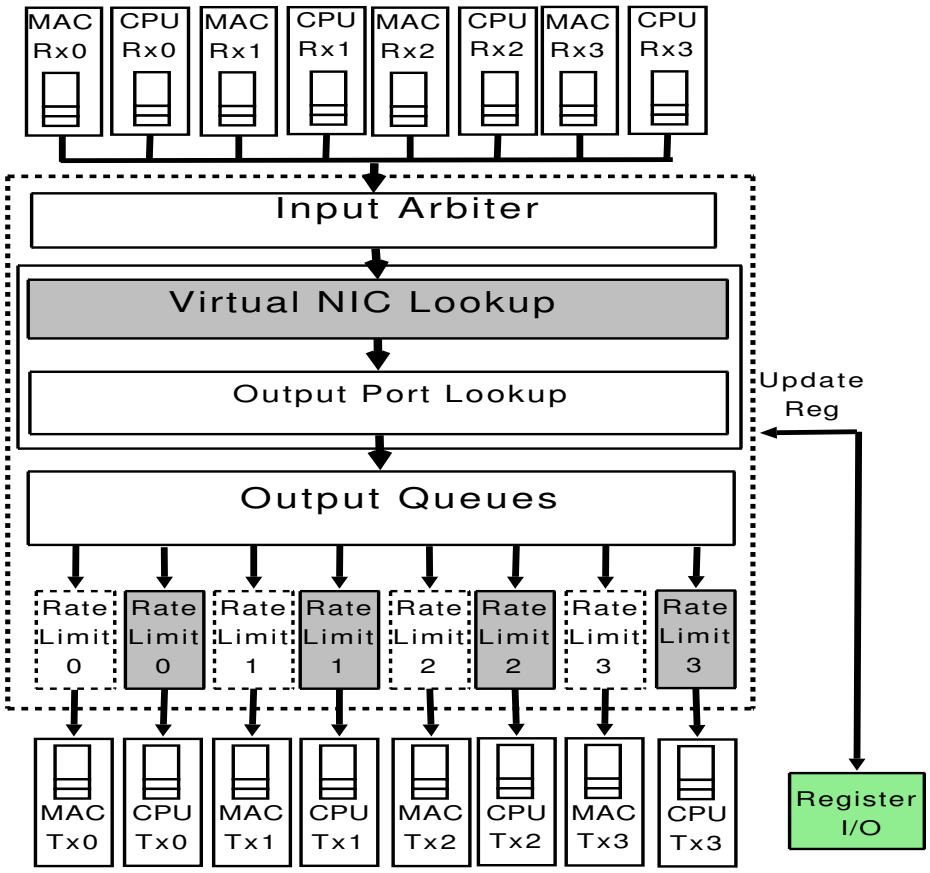


Figure 15: Pipeline for NetFPGA implementation.

4.4 Implementation

We implemented this virtual NIC design with a mapping table and rate limiters for receive-side fairness for virtual machines. Through this implementation, we wanted to show the feasibility of the design and its ability to maintain network I/O fairness and interrupt suppression.

Figure 15 shows our implementation on a NetFPGA [10] card, with the possibility of adding rate limiters on the transmit side. Figure 15 also shows the pipeline of the implementation with new modules for virtualized Ethernet card highlighted. We have used the NetFPGA reference implementation as the base implementation. We have combined "Output Port Lookup" and "Virtual NIC Lookup" stages into a single stage and have not implemented transmit-side rate limiters for the CPU queues which are

shown with dotted lines in figure 15.

There are four rate limiters in hardware to provide receive-side fairness for the incoming packets to all the virtual machines running in software. Similarly, transmit-side rate limiters can be used to stop any virtual machine from sending more than its fair share (Figure 15). Because the packet generation process is handled by the CPU and each virtual machine's CPU cycle scheduling is already being done by Xen VM scheduler, it should not be possible for a VM process to get more than its share of CPU cycles allotted by scheduler and then generate high volumes of traffic. However, a user might still send traffic from unauthorized physical Ethernet interface; to counter this, we have an entry in mapping table that keeps track of outgoing traffic and prohibits any user from sending at higher than the allotted rate.

We have implemented queue mapping using a single BlockRAM-based CAM with exact matching. There are 32 entries in CAM; each entry has 3 fields. For each entry, there is a single MAC address with administrator allocated incoming and outgoing ports for it to access.

The NIC matches the destination MAC address of each packet against the CAM; if there is a hit in the table, then the card sends the packet to the corresponding CPU queue, as determined by the values that are stored in the CAM. Figure 15 shows the CPU queues after the MAC lookup stages as "CPU TxN". If there is a miss, then the packet is dropped immediately in the "Virtual NIC Lookup" stage.

For packets coming from the CPU, the interface card looks up the packet's source MAC address; if there is an entry for the source MAC address, then the interface card knows that a legitimate user sent the packet. The interface card then identifies this user's outgoing port and the packet is sent out of the allocated physical port to the particular user.

Table 9: Resource utilization for the Virtualized NIC with four rate limiters

Resource	V2Pro 50 Utilization	% Utilization
Slices	15K out of 23,616	63%
Total 4-input LUTs	21.5K out of 47,232	45%
Route Thru	2K out of 47,232	4.3%
Flip Flops	15K out of 47,232	31%
Block RAMs	116 out of 232	50%
Eq. Gate Count	8,176 K	N/A

4.5 Results

In this section, we present the resource usage of an initial NetFPGA-based hardware implementation and the performance results.

4.5.1 Resource Usage

For the resource usage of our implementation we used Xilinx ISE 9.2 [161] for synthesis. The resource usage reported here is for a design with 32-entry CAM and with four virtual interfaces on the card. Adding more virtual Ethernet interfaces will mean adding more queues on the NetFPGA card which means more resource usage. Here we are using four rate limiters on each Ethernet queue for receive side fairness therefore we only need a four entry table.

Our existing implementation, uses 21,500 four-input Look-Up-Tables (LUTs) of the 47,232 available. This makes 4-input LUT utilization to be 45% of available resources on the Virtex-II Pro FPGA [159]. Table 9 provides various other statistics about resource usage. This resource usage will increase once we increase the number of queues in hardware and add rate limiters to those queues; it will decrease if we have less than 32 entries in the CAM.

4.5.2 Performance

We first show that our simple virtualized network card implementation works by assigning each virtual network interface a specific MAC address that can be used

by the virtual machine. Current implementation has four queues in hardware and four interfaces in the software. In this experiment, we simply show that what is the maximum amount of traffic that can be sent from the outside servers to the virtual machines on server. Although the PCIe interface has much higher bandwidth than PCI interface, the 1x4 Gbps NetFPGA [10] card only has PCI interface; therefore, we have done measurements on NetFPGA card connected to the server through PCI interface.

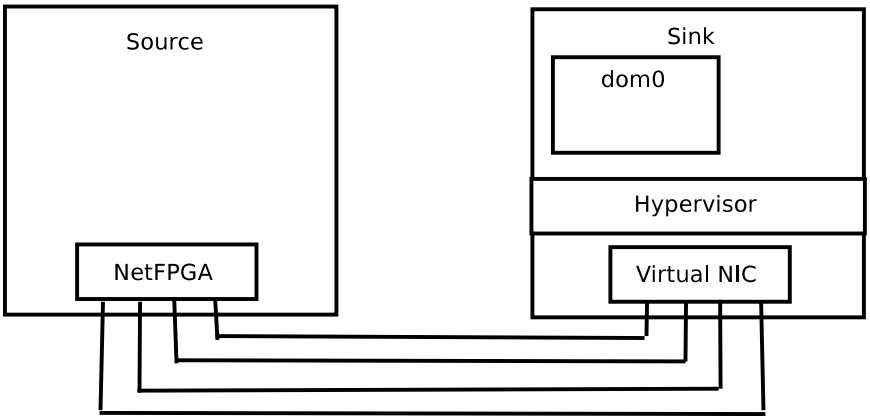


Figure 16: Experimental Setup

Figure 16 shows our experimental setup, which consists of 2 1U servers with Intel Quad Core processors and 4GB of RAM on each machine. Each machine has one NetFPGA card. It is a single source-sink topology where source is directly connected with the sink. All four ports of the source node are connected directly to the four ports of the sink node. We used the NetFPGA-based packet generator [51] to generate high speed network traffic. On the receiver side, we had NetFPGA card with virtualized network interface card, as discussed in Section 4.4.

4.5.2.1 Packet Demultiplexing

We measured packet-receive rates in the Xen hypervisor to obtain a baseline measurement for forwarding speeds of the PCI-based NetFPGA NIC card. Figure 17 shows total receive rate when the packet generator floods one, two, and four queues

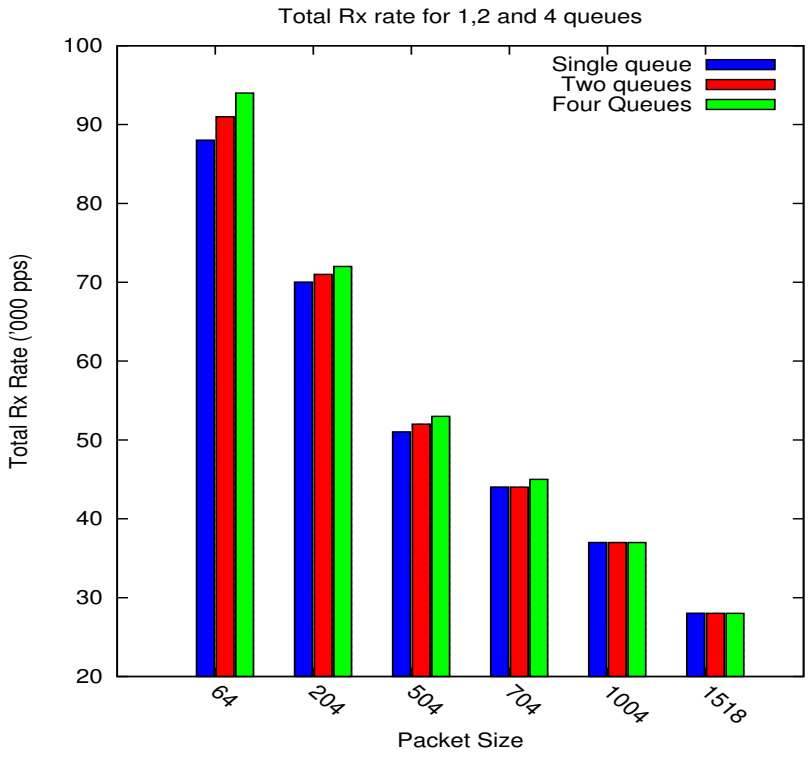


Figure 17: Total number of received packets per second at different packet sizes, for one, two and four queues

of the network interface card. As we increased the number of queues from one to four, the total packet receive rate remained the same, and the bandwidth was distributed evenly across different virtual machines. Interestingly, this flooding showed fairness in the number of packets received by more than one queues. When we flooded two and four queues, the cumulative rate remained the almost same, and all queues received packets at equal rate. This shows that the virtualized NICs achieve fairness even when all hardware-based rate limiters are disabled.

Figure 18 shows the per-queue receive rate in packets per second that can be achieved when the NetFPGA card is in the PCI slot. When we increase the number of receive queues that are being flooded, the forwarding rate per port is decreased: the forwarding rate is inversely proportional to the number of queues.

These two figures(Figures 18 and 19) show the inherent fairness in the virtualized network interface card. When all users are using the network at full capacity, traffic

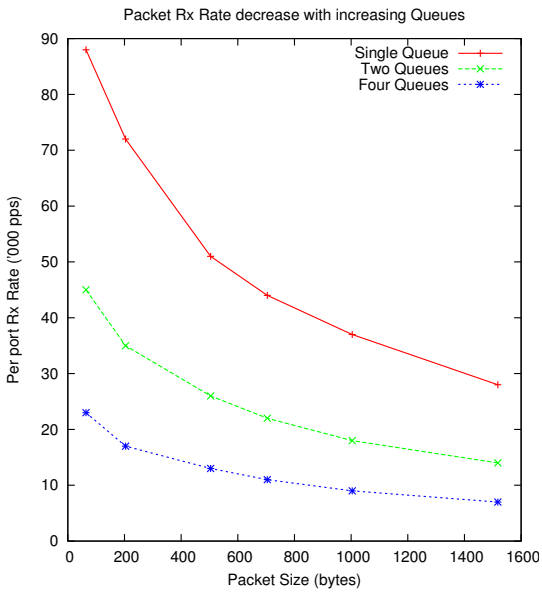


Figure 18: Number of received packets per port at different packet sizes

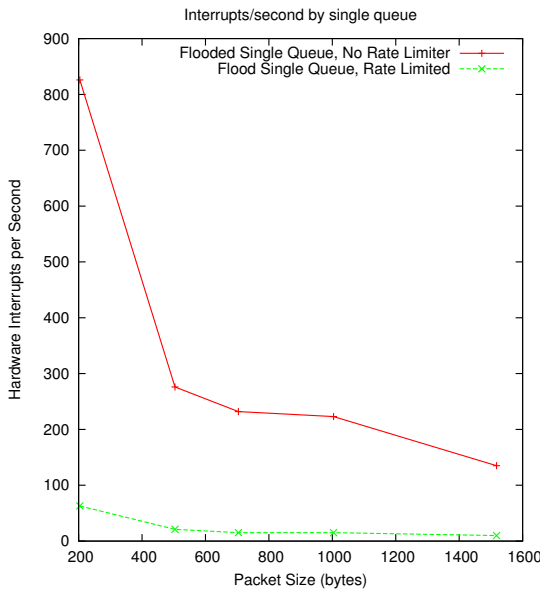


Figure 19: Reduction in Hardware Interrupts to CPU

is equally shared among them. We are able to get equal shares for all the users mainly because of virtualized queues, and by representing each queue in software as a separate network interface. It also shows that with the PCI 32-bit version working, we can achieve a maximum speed of about 90,000 packets per second with 64-byte sized packets. While each queue is receiving packets at approximately 23,500 packets per second.

4.5.2.2 Hardware Interrupt Suppression

As shown previously that in virtualized NIC cards, assigning separate queues to each VM, provides equal share of network traffic to each VM. But the problem comes in when we consider the number of interrupts sent to hypervisor because of traffic of each VM. If one of the four user is flooding the network and three users are using bandwidth within their limits the number of interrupts sent to hypervisor from user flooding the card will be much more higher then users staying within their limits. Here we show that by rate limiting in hardware we are effectively suppressing the user interrupts as well thus providing the hypervisor and dom0 ability to serve equal

amount of CPU resources to each VM.

To measure the effectiveness of this implementation we measured the decrease in interrupts per second by decrease in packets received by the Xen Hypervisor. We measured the interrupts in hardware using open source tool Oprofile [109].

Figure 19 shows the results for this particular experiment. Here we flooded a single virtual Ethernet card with maximum possible traffic to the card. For the smallest size packets the number of interrupts per second was almost equal to the number of packets per second received by the server.

As the packet size is increases from 64 bytes to 1518 bytes, the number of interrupts per second drops. This drop in interrupts per second does not guarantee fair resource sharing on the CPU secondly for smaller packet sizes number of interrupts per second is too large.

To measure the effectiveness of the rate limiter in decreasing the number of interrupts per second, we decreased the amount of traffic received by each user to approximately 24,000 packets per second for 64B sized packets. Then we flooded the single queue with NetFPGA packet generator at approximately 1Gbps with 64B sized packets. Despite this higher rate from the packet generator, the virtualized network interface card only forwarded packets that were allowed for the particular single queue. As shown in Figure 19, this directly resulted in a decrease in the number of interrupts per second to the hypervisor.

This essentially shows that a simple rate limiter can reduce the number of interrupts to the hypervisor and dom0 and can stop a user from taking unfair advantage of CPU resource in the hypervisor and domain0 for that user.

4.5.2.3 CPU Resources

To measure the effect of excessive packets on the CPU we measured number of CPU cycles required to process each packet and instructions taken by the CPU to process

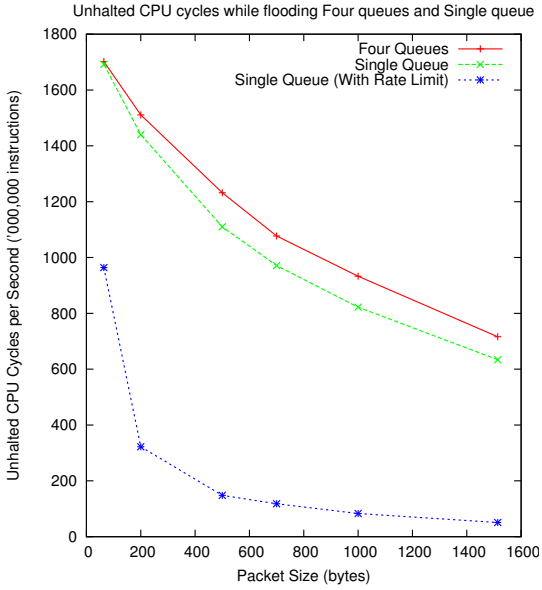


Figure 20: Unhalted CPU Cycles per second while flooding.

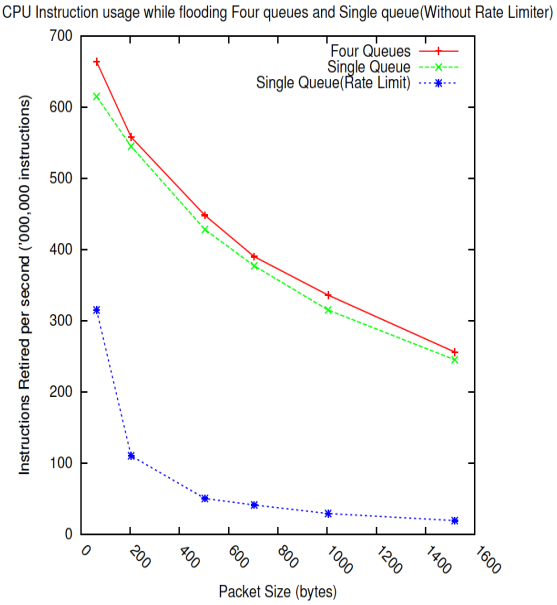


Figure 21: Instructions per second while flooding

those packets. For this experiment we used Xen and dom0 kernel with debug information, which resulted in overall lower packet capturing rate for the kernel. But the packet per second behavior was similar as shown in previous experiments.

We measured the number of unhalted CPU cycles on a 2.66 GHz Intel Xeon processor and number of instructions executed to capture the packets. First, we flooded all four ports of the network interface card with the network traffic and measured the number of unhalted CPU cycles per second and number of instructions executed per second, while all four queues were flooded. We repeated this process for different packet sizes.

As shown in Figure 18, the per-port receive rate decreases when we increase the number of queues being flooded. The same thing is happening here, number of CPU cycles to handle each machine's packets are distributed for different virtual machines. Figure 20 shows the number of CPU cycles being spent to serve the packets for a single queue without rate limiting, is almost equal to the number of cycles being spent when all four ports are getting equal network traffic share.

Then we limited the amount of traffic that can be sent using single virtual queue equal to 24,000 packets per second for 64-byte packets using rate limiters. After putting a limit on what traffic can be sent to a single virtual machine using the rate limiters, the number of cycles spent to serve a single queue's traffic decreased. Apart from the smallest packets, we see more than four times decrease in number of unhalted CPU cycles for larger packet sizes.

We repeated this same experiment and measured the number of instructions per second. The number of instructions spent per second with single queue being flooded was almost equal to when all four queues were bombarded using the packet generator, as shown in figure 21.

After enabling the rate limiter, we reduced the single queue's forwarding rate. Figure 21 shows the results of enabling the rate limiter and setting a lower rate limit on a single queue. We observe a decrease in the instructions per second spent by the Xen Hypervisor and dom0 for a single virtual machine user that is more than a factor of four, for all packet sizes except 64-byte packets.

These experiments show two benefits of locating rate limiters in the virtual network interface card itself. First, using rate limiters, we can send through only that traffic to the server and to the virtual machines that they can handle in software, without excessively increasing the number of unhalted CPU cycles and instructions spent on the packets.

Second, rate limiters in the virtual network interface cards do a nice job of stopping the hypervisor and the driver domain in spending extra CPU cycles for unwanted traffic. Although the Linux traffic shaper can be used in dom0 to limit traffic to domU, using this technique will still mean that hypervisor and dom0 must receive and process the packets before discarding them. Using rate limiters can handle such unpredictability in hardware, thus allowing the Xen scheduler to more efficiently schedule the VMs themselves.

4.6 Summary

Operating system virtualization is appears in both cloud services [24] and router virtualization [58]. Until recently, however, network virtualization for virtual machines occurred entirely in software. Recent solutions achieve better performance by using hardware to multiplex and demultiplex the packets, before sending them to operating system. Moreover, network virtualization for operating systems means following existing models that have proven to be useful, such as providing fault isolation with a separate driver domain.

This chapter takes a step towards providing a fast, flexible, and programmable virtualized network interface card that can provide high-speed network I/O to virtual machines while maintaining the driver domain model. In addition to providing packet multiplexing and demultiplexing in hardware for virtual machines, assigning each virtual machine to a single queue in hardware and using the network traffic rate limiters in hardware achieves network I/O fairness and maintains good performance by suppressing interrupts in hardware. This hardware-based approach reduces the interrupts that are sent to the hypervisor, as well as the number of instructions and number of CPU clock cycles spent to process each packet. Although many more functions can be added to the proposed virtual network interface card, this chapter represents a first step towards providing hardware-assisted network I/O fairness for virtual machine environments.

CHAPTER 5

HETEROGENEOUS SWITCH

5.1 Introduction

Software defined networking (SDN) paints a vision of highly programmable network devices that can forward traffic at line rate, but the reality is that most programmable network devices still operate either largely in software (and, hence, cannot forward packets quickly) or on hardware platforms that offer a narrow programming interface but have only limited programmability. The most prominent software defined networking paradigm to date is OpenFlow [117], whereby a software controller can effect a set of forwarding actions based on a set of characteristics of packets belonging to a traffic flow. Unfortunately, both the set of actions that these switches can perform and the set of flow characteristics that these switches can match are limited. These limited set of flows and actions restrict the functions that a network operator might ultimately want to perform.

The diversity of packet-processing hardware—ranging from ASICs and FPGAs to network processors and CPUs—presents the necessary underlying machinery for fast, custom processing of network traffic as proposed in previous work [71, 75, 102, 103, 140]. Unfortunately, however, we have no common substrate that allows network operators to apply and compose these custom packet processing elements into a single, customizable data plane.

This chapter presents LEGO, a programmable switch abstraction that allows network operators to implement custom forwarding operations(Chapter 3), network functions/elements(Chapter 6) and performance enhancements(Chapter 4) by defining

flexible data paths through custom packet processors that they also program. Current OpenFlow switches provide only a limited notion of flows and actions; LEGO expands both, the sets of conditions on which traffic forwarding decisions can be made and the set of actions that a switch can take based on those conditions.

LEGO integrates heterogeneous custom packet processing units (*e.g.*, programmable hardware) into a common switched backplane, allowing network operators to specify network policies based on a much broader range of flows and actions. LEGO defines groups of traffic flows not only on the conventional fields that are used to define OpenFlow rules, but also on additional (and custom) properties of packets—this extension makes it far easier to define forwarding conditions based on emerging header formats or fields (*e.g.*, regular expressions in packets [67]). Operators can also define actions much more broadly: for example, in addition to conventional OpenFlow actions such as dropping and forwarding traffic, LEGO allows more complicated actions such as rate limiting, packet rewriting [145], encoding, and compression [67].

Realizing LEGO involved tackling three challenges. The first challenge is designing the *hardware configuration* for the LEGO switch itself to allow high throughput and low latency and to permit a simple programming model. Our design for LEGO essentially boils down to turning a switch "inside out": We use an OpenFlow-enabled switch as the backplane between custom packet processing modules. The second challenge is developing a *device abstraction* for programming the heterogeneous custom packet processing units (*e.g.*, NPUs, FPGAs) using a single abstraction on custom packet processor, Our device abstraction results in developing LEGO Runtime and LEGO controller that can be used to discover custom packet processors attached to a server and convert them into peripherals of switch for expanding its conditions and actions. The third challenge is designing a *data-plane abstraction* that allows network operators to integrate custom packet processors' functions with OpenFlow switches that "stitch together" custom packet processing pipelines for traffic flows. LEGO uses

the OpenFlow control protocol to install forwarding table entries in the backplane switch that in turn dictates specific packet-processing pipelines through a sequence of custom packet processors, based on the properties of each flow.

In principle, LEGO can incorporate any custom packet processor that exposes a control interface via either PCI or Ethernet. In our prototype implementation of LEGO, we used NetFPGA interface cards as example custom packet processors and show how the LEGO framework can be used to specify a custom packet processing pipeline using these custom packet processors. In principle, however, LEGO design is more general, and could incorporate other packet processing devices, such as NPUs [3, 67, 85]. Our evaluation shows that LEGO achieves latency and throughput that is comparable to state of the art OpenFlow switches, while permitting a much wider range of flow conditions and actions than existing switches.

The rest of the chapter proceeds as follows. Section 5.2 presents the LEGO design and Section 5.3 presents our implementation of LEGO, which uses an OpenFlow switch as the backplane and NetFPGA interface cards as the custom packet processors. Section 5.4 evaluates LEGO's performance for a variety of applications, and Section 5.5 concludes with a summary of contributions made in this chapter.

5.2 LEGO Design

We now describe the LEGO design; we provide an overview of the design and then proceed to describe each component in detail.

5.2.1 Overview

LEGO has four components:

- *Custom Packet Processors (CPP)* are packet processors that are attached to an OpenFlow-enabled packet forwarding device through Ethernet interfaces. CPPs are used to expand the "flow" definitions or "action" sets beyond what is

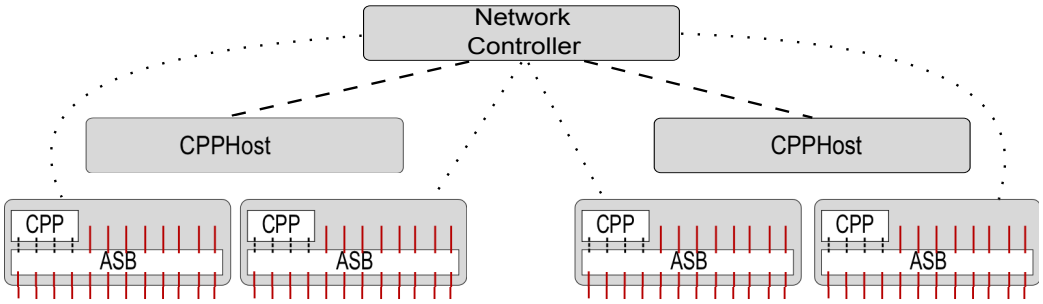


Figure 22: LEGO design configuration. The dashed line between network controller and CPPHost, represents a connection from each LEGO Runtime to the controller, we call it LEGO control channel. The dotted lines represent OpenFlow control channels. The dashed lines from ASB to CPP and dashed lines between controller and CPPHost are links specific to LEGO.

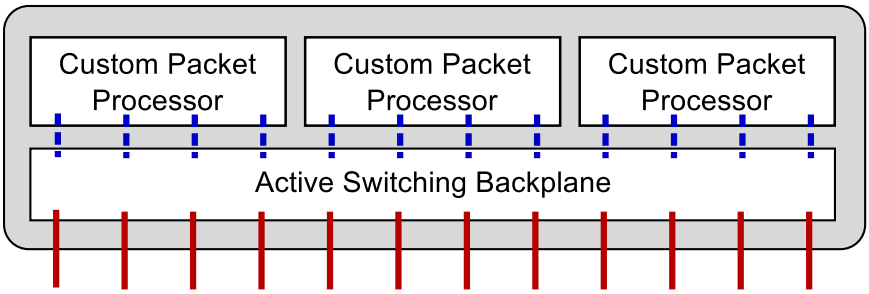


Figure 23: LEGO configuration with a 24-port active switching backplane. The custom packet processors connect to the core via Ethernet links. Solid lines are Ethernet links for external traffic; dashed lines are Ethernet links for internal traffic.

possible in today's SDN switches. (§5.2.2) Section 5.2.3 describes how LEGO defines abstractions for these CPPs.

- The *LEGO Controller:* a controller application that installs flow table rules to control the behavior of the active switching backplane and install rules on the custom packet processors using LEGO Runtime.(§5.2.4)
- *The LEGO Runtime* hosts the custom packet processors and updates the logic on these CPPs. (§5.2.4)
- The *Active Switching Backplane (ASB)* implements the programmable switching fabric that implements an interface such as OpenFlow and can be programmed through a network controller. (§5.2.5)

Figure 22 shows the LEGO design; the dashed line links between CPPHost and

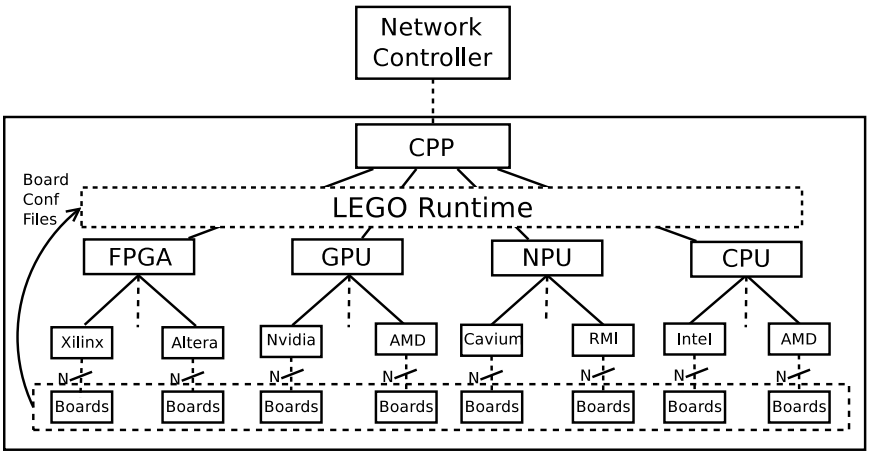


Figure 24: LEGO Device Abstraction: LEGO abstracts away the complexity of different packet forwarding devices and presents them as a simple CPP to network controller.

Network Controller depict control channels that LEGO adds. Each LEGO Runtime has a communication channel with the Controller. This channel performs the initial boot process of the LEGO Runtime and is the conduit for any control information between the controller and the CPPs that is communicated through LEGO Runtime. One LEGO Runtime can connect to more than one active switching backplane, and each active switching backplane can have more than one CPPs. In the basic configuration, a CPP's ports connect to the ASB, and all traffic from external sources enters and exits through the ASB, as shown in Figure 23. In this configuration, the CPPs act as custom traffic processors; when the ASB receives traffic from a host, it passes the packets to and from the CPPs and switches the processed traffic to its destination. This design allows LEGO CPPs to augment existing network switches.

LEGO's *active switching backplane* is a low-latency, high-bandwidth programmable switch; the ASB decides how to forward a packet (*e.g.*, to an output port or a processing element). *Custom packet processors* (CPPs) connect to this core and implement custom processing and forwarding functions that the ASB cannot implement (*e.g.*, encryption, deep packet inspection). Figure 22 shows the design of LEGO and how the CPPs connect to the ASBs. The core can be programmed from software, offering flexibility and fast forwarding. The flexibility of the ASB provides loose coupling

between the CPPs, enabling the synthesis of heterogeneous CPPs with different programming models. In the remainder of this section, we describe these components in more detail.

5.2.2 Custom Packet Processors

A custom packet processor is a computational engine that is attached to an active switching backplane using an Ethernet interface to process any traffic coming from the ASB ports. As in RFC 3234 [40], we have classified CPPs according to two types:

- **Forwarding CPPs** modify the packet content so that its ultimate destination while going out of active switching backplane is changed. e.g. IPv4,IPv6 A *forwarding* CPP allows the network operator to *expand* or *change* the flow definitions defined by active switching backplane above layer 2 headers.
- **Processing CPPs** do not make any decisions on packet's final destination, but may or may not modify packet's content (*e.g.*, encryption modifies the packet, but rate limiting does not). A *processing* CPP can expand the set of *actions* that can be taken on the packets, these actions can include traffic rate limiting,traffic encryption, traffic compression etc.

A *Forwarding* CPP can also change a packet's contents, but a *Processing* CPP cannot change the packet's final destination.

5.2.3 Device Abstraction

A custom packet processor is a simple network interface card that can process the packets and return them to the host machine. A CPP has four main parts: (1) processing elements; (2) memory; (3) one or more Ethernet interfaces; and (4) a control interface (*e.g.*, PCI/PCIe). LEGO accommodates CPPs with all these parts on the same board to enable processing packets without sending them to the host server's CPU. A CPP can be standalone (*e.g.*, a PCIe-based packet processor cards) [4, 13]

or a combination of a network interface card and a CPU/GPU. A CPP processing element can be a simple processor(CPU) (*e.g.*, an Intel Core i7 processor or AMD Opteron Processor), a network processor (*e.g.*, [3,85]), an FPGA (*e.g.*, [158,159,162]) and a graphics processing engine [2, 11]. Within a single class of processing elements there are different architectures available; for example, in the case of simple processors, there are x86, MIPS, and PowerPC processors. Similarly, network processors include MIPS [3] and XScale architectures [85].

Each processing element has a programming model that exposes a different set of programming abstractions. FPGAs are programmed using a hardware description language and accessed through a register interface. NPUs, GPUs, and CPUs all provide respective libraries and SDKs (*e.g.*, [53, 116]) for accessing these processing elements, thus raising the level of abstraction and facilitating programming. Different CPPs also have distinct available hardware resources. A CPP can have a different number Ethernet interfaces. The LEGO Runtime hides all this complexity with the one CPP abstraction as shown in figure 24.

CPPs are normally placed inside a server and appear as server peripheral [4,10,13]. LEGO uses an approach where to program these "peripheral" devices from a central controller; it changes the status of these devices from "peripherals of a server" to "peripherals of a switch". This change of status requires identifying the location of CPP (*i.e.*, the switch connected to CPP) and creating unique identifiers so that the peripherals that were originally addressed through a computer bus can now be addressed from a network controller. LEGO creates unique identifiers that can be used to locate and address individual CPPs. To achieve this level of device abstraction and to make CPPs addressable at the network level, we developed the LEGO Runtime.

5.2.4 LEGO Runtime

The LEGO Runtime manages the CPPs that are connected via the LEGO active switching backplane. The LEGO Runtime runs on a server and uses a control interface such as PCI/PCIe to all of the CPPs that it controls.

The LEGO Runtime has three responsibilities:

- *Boot up and discovery.* The runtime boots the CPPs and discovers the CPPs that are connected to one of the active switching backplanes.
- *Control of CPPs.* The runtime manages the CPP control interface (*e.g.*, PCIe) to invoke functions on the CPP designs.
- *Programmatic interface.* The runtime provides an interface to the controller to program the CPPs.
- *Device and board configuration.* The runtime interacts with CPP device configuration to help the LEGO Runtime determine the properties of the attached CPPs.

We now describe these tasks in more detail.

5.2.4.1 Boot Up and Discovery

The LEGO Runtime performs functions on CPPs on behalf of the centralized network controller (*e.g.*, an OpenFlow network controller). All of the custom packet processors must provide a control interface (*i.e.*, a host driver) to the Linux operating system that can be used to download and start any program on each type of CPP attached to the active switching backplane.

To boot custom packet processors, LEGO assumes that the devices are without any pre-programmed logic. It also assumes that there are no rules installed on the active switching backplane. Once the LEGO Runtime starts, one of its first jobs is

to detect the peripherals attached to a server and figure out their Ethernet interfaces and MAC addresses. It then downloads a program on each CPP that can send packets on behalf of LEGO Runtime to active switching backplane.

Once this download completes, LEGO Runtime establishes all of the interfaces for the CPPs and performs link detection to determine how many of CPP interfaces are connected to the ASB. Once link detection completes, the LEGO Runtime updates its record for each CPP about its connected ports. All of this happens before the "discover" process begins, as shown in Figure 25.

The first step in Figure 25 shows the *port discovery* process. After the network interface card logic is downloaded onto the CPPs that are attached to the active switching backplane, one "discover probe" is sent from each Ethernet interface of each CPP hosted on LEGO Runtime to the network controller, as shown with the "discover probe" line in Figure 25. The LEGO Runtime also sends a "discover" message to the controller. This message tells the controller that the LEGO Runtime is sending a packet from a CPP's interface with a specific MAC address asking for the active switching backplane's port number. The controller replies back on the LEGO control channel; it sends the port number to which the CPP's MAC address is attached and with active switching backplane's MAC address.

This process continues for all ports of all the CPPs attached to the LEGO Runtime. Once the port discovery process is done, the controller updates the CPP IDs. Each custom packet processor ID consists of a 48-bit active switching backplane MAC address and 16 bits for each port of the CPP.Where first 48-bits represent the MAC address of ASB and each 16-bit value represents the port number to which the CPP is connected with the active switching backplane. This results in a unique address for each port on all the ASBs which can be addressed by the controller individually.

5.2.4.2 Control of CPPs

There are four categories of CPPs: NPU (Network Processing Unit), GPU (Graphics Processing Unit), CPU (Central Processing Unit), and FPGA (Field Programmable Gate Arrays). Some CPPs might involve a combination of the above, (*e.g.*, a CPU with an FPGA [6]).

This division of CPPs into four categories is used for structuring code, as it allows LEGO Runtime to identify the right level of abstraction for the appropriate CPP. This abstraction dictates whether to issue a lookup for a specific SDK of a network processor, use a management library [12], or use the Linux kernel API. LEGO Runtime handles the control interface for different kinds of CPPs that it hosts. Different CPPs might provide different abstractions for implementing this functionality. Network processors usually provide a C API to program the network adapter; on the other hand, FPGA-based adapters for a specific design might only have a thin register interface that needs to be programmed by the individual programmer.

The LEGO Runtime dispatches function calls or register reads and writes on the individual CPPs on behalf of the controller. Until the boot process completes, the control interface is used by LEGO Runtime for bringing up the individual CPP interfaces. This interface is also used for discovering CPP's Ethernet interface attachments to ASB ports. Once discovery completes, this interface performs only those actions that are allowed by the LEGO controller application.

5.2.4.3 Programmatic Interface to Controller

Once the port discovery process is complete, the LEGO Runtime subscribes to the controller and waits for instructions. The LEGO Runtime exposes a set of functions that are invoked by the controller to perform different operations on the CPPs hosted on the LEGO Runtime. The controller uses active switching backplane ID and CPPID to locate the CPP hosted on the LEGO Runtime, implementation of a function is

dependent on individual CPP designs. Step 2 in Figure 25 shows how any function invocation on CPPs requires the controller to send the request to LEGO Runtime, which dispatches it to CPP through its control interface.

Rule installation is a two-step process: it requires installing rules in the ASB as well as installing the rule on the CPP. Every flow that requires specialized packet processing requires three rules. Two of these rules are installed on the ASB for outgoing and incoming traffic from CPP; one such rule is installed on CPP. This 3-step rule installation requires installation of all three rules before declaring a rule as installed.

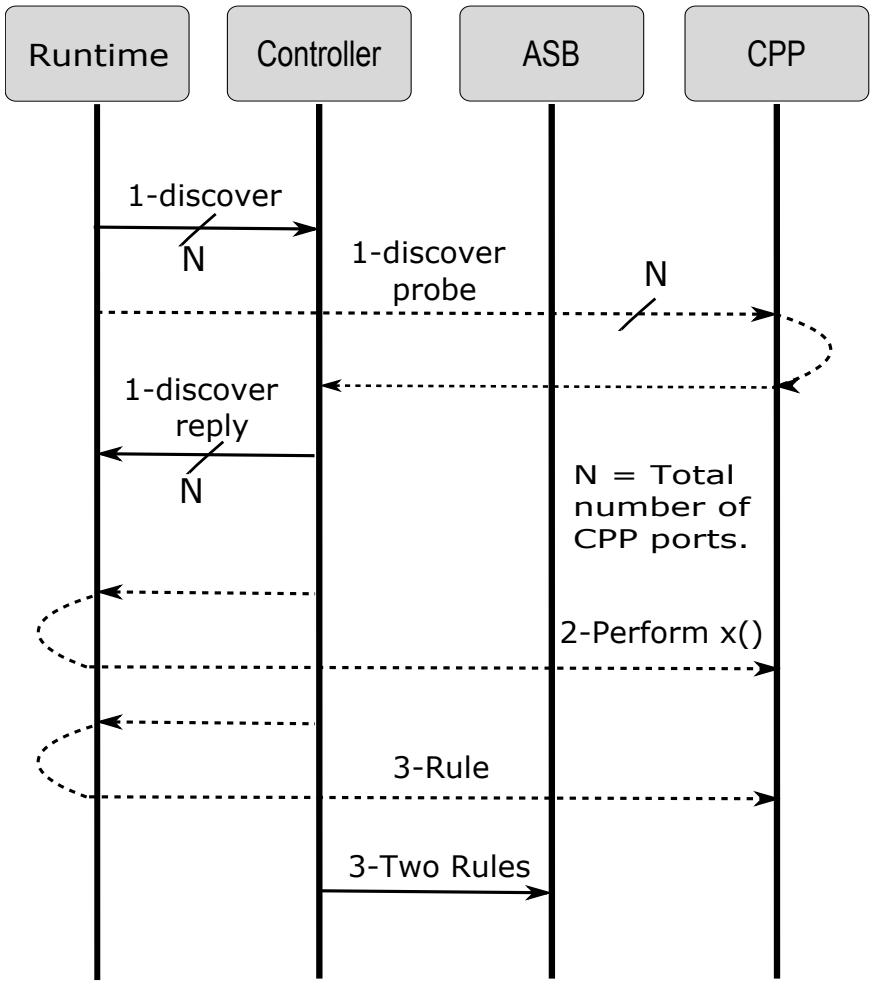


Figure 25: LEGO modules communication. An arrow-head represents the path taken by the packets.

5.2.4.4 Device and Board Configuration

Board configuration files specify the properties of a CPP for LEGO Runtime. These properties are used by LEGO Runtime to probe a CPP to make it a switch peripheral. The concept of device configuration files is borrowed from debuggers [5], which have a huge spectrum of embedded processors and their motherboards to support. These processors have unique properties including, different register files, different cache size, and different debugging ports. Similarly, a single processor can be hosted on motherboards of different configurations, where the memory and peripherals connected on the board can be of different sizes and different types. Given the diversity of platforms and functionalities, LEGO uses a similar approach to device configuration files to identify the CPPs attached with a server.

Each new board that is added to the set of boards supported by LEGO requires to specify its name and version. The name and board version, should be specified such that it should allow us to uniquely identify the CPP among different types of CPPs available. We have to tell the type of the processing element, if its an FPGA,CPU,GPU etc. This type specification is important as it allows LEGO to put the board into one of the four categories of FPGA,CPU,GPU,NPU. Thus allowing it to make assumption about the level of abstraction provided by the board for code and status update. For example if the device type is FPGA then LEGO knows that it should be looking for the design's interface files instead of assuming an installation of Linux OS.

Board configuration file has Ethernet ports present on the board and their bandwidths. The current implementation assumes that a board will have all Ethernet ports of the same bandwidth 1G/10G/40G. Another important field of board configuration file is the signature of the board as it appears on a Linux server. This signature involves the class of the device, its vendor, its device name assigned by vendor and the interface name (*e.g.*, eth*, nf2c*, octeth*) as it appears on the Linux

server. The signature on a Linux server helps LEGO to identify individual boards attached on the PCI/PCIe slot of a server. These signatures act as an identifier for the device at the server level until the network controller assigns the device a unique identifier.

These files also contain information about the tools specific to a board and the parameters to use for a particular board. This information is needed for each board as different boards that have the same processing element might require different tools to download code on them. Similarly, this file also has boot parameters to be provided to start executing the program on the processing element.

5.2.5 Active Switching Backplane

This section discusses the design of LEGO's active switching backplane, which is an active switching fabric that makes forwarding decisions based on the *contents* of the packet header and instructions from the ASB controller. decisions represent standard switch and Unlike a conventional switch, however, some of the forwarding decisions may entail sending traffic to a CPP, rather than directly to an output port. The LEGO Runtime provides an abstraction at the control plane for addressing both *forwarding* and *processing* CPPs. Supporting forwarding CPPs requires an abstraction for the data-planes to allow CPPs relay forwarding decisions to ASB at high speed. Using an ASB presents a unique opportunity, where a combination of hardware and software CPPs can be coupled to form chains (§ 5.2.5.2). Thus, support for forwarding CPPs combined with chaining required an abstraction that could be used to communicate between LEGO's data-plane elements; we present this abstraction below.

5.2.5.1 Communicating with the CPPs

Custom packet processors can communicate with the active switched backplane or with each other by passing state through LEGO header. Figure 26 shows the header

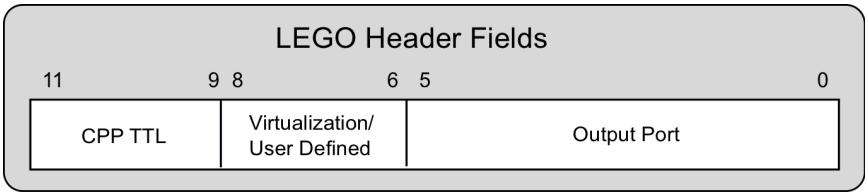


Figure 26: The LEGO header co-opts the twelve bits of the VLAN header. The first six bits set the output port for the packet. The next three bits are *user-defined and can provide instructions for the CPP or as output port for switches supporting up to* 512 ports *[7] The last three are used to enable chaining.*

Table 10: LEGO header support requirement for different types of custom packet processors.

CPP Type	Chaining	LEGO Header
Processing	No	No
Forwarding	No	Yes
Processing	Yes	Yes
Forwarding	Yes	Yes

that each packet should carry inside the switch if it is to be treated by a forwarding CPP or a CPP chain. The header contains instructions for the backplane that help it direct packets to the appropriate packet processor or to the output port.

LEGO uses the standard VLAN header to communicate between the custom packet processors and the active switching backplane. Our decision to use the VLAN ID for the LEGO header rests on two observations. First, we wanted to use a standard layer-2 technology that is widely used in today's networks. Second, the emergence of the VxLAN [155] header combined with existing support for VLAN headers provides reasonable flexibility for using VLAN header bits as the LEGO header.

Table 10 shows the changes required to support different functions in CPPs. If required packet processing does not require chaining CPPs and is only performing processing, then the CPP can be integrated with active switching backplane without any changes. On the other hand, if the CPP makes any forwarding decisions or it needs to be a part of a chain, then it must be able to understand the LEGO header.

The ASB redirects the traffic that requires on expanded definitions of "flows" or new set of "actions" that the active switching backplane currently cannot support (*i.e.*, because it only supports standard OpenFlow rules and actions). For example, with OpenFlow as the ASB, if a host needs support for custom packet forwarding at layer 3 or, say, traffic compression, the ASB can redirect packets originating from the host to the CPP that can provide the appropriate function.

A *Forwarding* CPP modifies the LEGO header and writes its output port decision in "Output Port" field in the LEGO header. Once the "Output Port" field is modified, the ASB can forward the packet on wards. The interpretation of output port bits will depend on the CPP TTL bits, as shown in Figure 26. If these bits are zero, then the active switching backplane will send the packets out of the switch; otherwise, the ASB will send the packets to another CPP for further processing, based on the installed flow-table entries.

5.2.5.2 Chaining

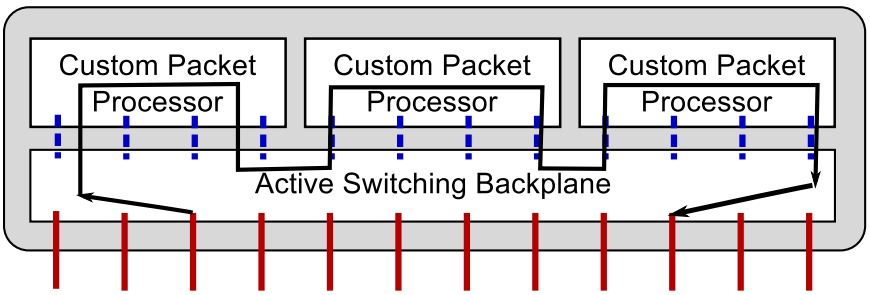


Figure 27: LEGO chaining: Paths taken by a packet requiring three CPPs.

Chaining allows a network operator to concatenate multiple custom packet processors without requiring any physical or hardware changes to the active switching backplane. Figure 27 shows an example of chaining. To enable flexible pipelines in the data plane, chaining allows LEGO to compose multiple *Forwarding* or *Processing* CPPs into a single logical custom packet processor. Chaining is useful when the processing logic or state for processing does not fit onto a single CPP. LEGO can compose several

custom packet processors in a single packet processing chain that creates a single *logical* custom packet processor.

Chaining can also distribute *state* across custom packet processors when a particular device may not have the necessary amount of state to process all traffic flows (*e.g.*, if all forwarding-table entries for a system cannot fit onto a single forwarding device).

Chaining offers one way to provide *loose coupling* between custom packet processors. An alternative to chaining is to use consolidated software functionality in-CPPs [133], but such an approach requires software implementations of each respective packet processing functions. In contrast, LEGO's custom packet processors can have radically different hardware specifications, programming models, and modes of interconnection; chaining helps a network operator compose these devices using the active switching backplane's interface (*e.g.*, OpenFlow) as the programming abstraction, rather than requiring the programmer to port software to different platforms that runs across different types of hardware.

One of the main drawbacks of chaining is that it wastes bandwidth at the active switching backplane, since traffic must traverse the ASB multiple times as it passes through multiple CPPs. It also requires two rules for each flow to use a single CPP; one for traffic going towards CPP and one for traffic coming out. When installing rules, a network operator must take care to avoid introducing unnecessary loops or other inefficient uses of the ASB's bandwidth. Chaining also increases latency; fortunately, in Section 5.4, we show these latencies that result from chaining are comparable to those provided by other programmable router platforms.

5.2.6 How to Program LEGO

We describe how to program LEGO, and how new *devices* and *designs* can be incorporated into LEGO.

The LEGO Runtime can detect NPU, GPU, FPGA, and other proprietary hardware. These peripherals are discovered and registered with controller as switch peripherals. The process of adding a new functionality on the data path starts by requesting a *design*, which we define as a program executable with an interface to run it, configure it, and to update its status. LEGO uses the MAC addresses of the respective active switching backplane ports as the data path ID. If there are more than one CPPs available on the given data path ID and the requested design can be downloaded on any one of those CPPs, LEGO reserves a CPP for the requested design and the CPP ID, which the LEGO Runtime returns to the programmer. Once a CPP is reserved, the next step is to download the requested design and configure it. A network administrator does not need to know the specific tools to download the design on a specific architecture; with help from LEGO Runtime the design is downloaded. The administrator provides the appropriate configuration parameters.

A CPP may be available for a data path even though the requested design cannot be downloaded on the CPP. For example, if the LEGO design library only has an IPSec implementation for x86 processor, then the implementation cannot be downloaded on a CPP with MIPS processor; in such cases CPP is not reserved for the design.

After configuring the design, forwarding and processing rules can be installed or removed on the CPP. Here the administrator needs to know the semantics of the rule to be installed. For example, in case of IPv4 router, a rule involves providing the destination IP address, output port, and next hop's ARP table entry. In case of a cryptographic NIC, the key is installed for traffic encryption/decryption and no further rule updates are required. As shown in step three of Figure 25, rule installation requires the administrator to install three rules, two rules on the active switching backplane and one rule on CPP.

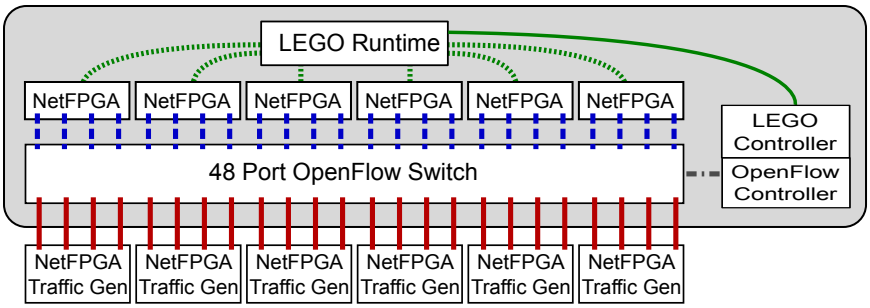


Figure 28: Schematic of LEGO implementation. The solid lines between traffic generator and switch are external Ethernet links, dashed lines are internal Ethernet links to CPPs, the dotted-dashed line is the OpenFlow control link, and the dotted lines are the CPP control interface over PCI. The solid line between LEGO Runtime and LEGO controller is LEGO Runtime control.

Adding new CPPs. Adding a new CPP requires creating a board configuration file that describes the CPP. The device must have host drivers installed on the server to be recognized by the Linux kernel. Different CPPs can have different tools to download code and update their status, but these tools must be installed on the server or be part of LEGO software package.

Adding a new design. Adding a new design requires implementing a basic LEGO design abstraction for a particular design. This abstraction currently has five functions: acquire design, configure design, release design, install rule and delete rule. Implementing these functions can require defining specific files to program the CPP. For example, adding the IPv4 router design to the LEGO Runtime requires the header files that provide a mapping between register names and their mapped addresses.

5.3 Implementation

LEGO's implementation uses an OpenFlow-capable switch as the active switching backplane and an OpenFlow controller as the ASB controller. Although our implementation is based on 1G Ethernet technology, LEGO's design could also apply to a 10G or 40G configuration. LEGO's implementation has three parts: (1) LEGO's hardware configuration, (2) LEGO Runtime and LEGO Controller; and (3) Hardware

Design.

5.3.1 Hardware Configuration

Figure 28 shows the schematic, and Figure 29 shows our LEGO prototype and its evaluation framework, which has twelve 4x1G NetFPGA cards attached to the PCI bus, which is in turn connected via a PCI card to a 1U server. The prototype uses a 48x1G Pronto 3290 switch with OpenFlow firmware for the active switching backplane and six NetFPGA 1G cards as custom packet processors in closed port configuration to process traffic; six additional NetFPGA cards serve as traffic sources and sinks. OpenFlow allows us to separate the data and control planes in line with current network management trends and to implement various packet forwarding and manipulation operations that depend on being able to match fields in the packet header.

We place all the NetFPGA cards that act as custom packet processors in a Magma 13-slot PCI expansion system [105] that is connected to a 1U server that hosts quad-port Gigabit Ethernet cards. The server has an Intel Xeon 5600 processor, 1333Mhz of FSB, and 24 GB of RAM. It has a PCI card that enables it to connect with the PCI expansion system, as shown in Figure 29. All of the NetFPGA cards appear as PCI peripherals and can be programmed directly. LEGO forwards all traffic directly on the NetFPGA; we only use the PCI interface to the NetFPGA cards to read/write register values, or to send commands to reload any new logic on the NetFPGA cards. This 1U server combined with the PCI expansion system acts as the LEGO Runtime.

LEGO Runtime and Controller The server runs the LEGO Runtime that is responsible for the boot up and discovery protocol and provides the controller an interface to program CPPs. The LEGO controller code is implemented as a NOX [114] application and runs on an OpenFlow controller.

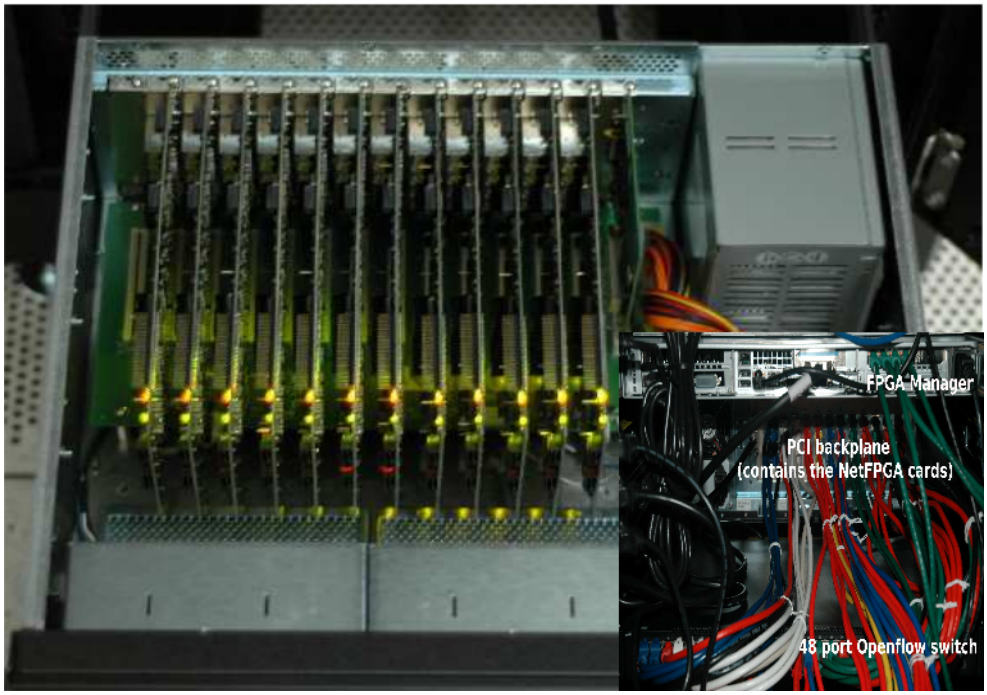


Figure 29: LEGO prototype, showing 12 NetFPGA cards connected to a PCI expansion system, which is connected via a PCI card to a 1U server. Inset shows the backside of the setup. On top is the server running LEGO Runtime; in the middle is a 4U PCI backplane; at the bottom, there is 48-port 1G OpenFlow switch.

5.3.2 Hardware Design

We implemented two different custom packet-processing functions on the NetFPGA cards: (1) CryptoNIC(an example of *Processing CPP*), which encrypts traffic at line rates; (2) an IPv4 router(an example of *Forwarding CPP*). We modified these designs to support LEGO header. We refer to the modified versions of these designs as CryptoNIC-L and Reference Router-L. We describe each of these implementations below.

CryptoNIC-L We modified the CryptoNIC implementation provided in the NetFPGA verilog repository to operate with LEGO. CryptoNIC-L(CryptoNIC for LEGO) is a processing element that performs per-packet operations but does not make any

forwarding decisions. We made only minimal modifications to original code of CryptoNIC to support VLAN tag manipulation.

Reference Router-L We modified the NetFPGA-based IPv4 reference router and added the ability to read and write to the VLAN field in the Ethernet header using NetFPGA verilog repository.

5.4 Evaluation

We now evaluate the feasibility of LEGO from different aspects. In §5.4.1, we measure the control plane latency introduced by using LEGO and overhead of installing rules on an Openflow switch with a single CPP. Then we look at the feasibility of LEGO's data plane latencies in §5.4.2 and compare existing technology with an implementation based on LEGO. In §5.4.3, we evaluate LEGO's ability to support twelve CPPs and measure overall system's throughput and individual CPPs throughput with and without LEGO header. Finally, we show the feasibility of chaining in terms of throughput and latency, respectively.

5.4.1 Latency of LEGO Control Plane

This section compares OpenFlow and LEGO rule installation latency. For OpenFlow, we use the *barrier* command, which causes the switch to notify the controller after insertion of a flow into the forwarding table; for LEGO we use synchronous communication between the OpenFlow controller and LEGO Runtime to be certain about rule installation completion.

To evaluate LEGO's rule insertion latency, we benchmark the flow insertion latency of a Pronto 3290 OpenFlow switch (with Indigo firmware) without installing any CPP rules. We used *pktgen* to generate 100 flows per second, each with one packet. Figure 30 shows these flow installation latencies for varying numbers of rules. Next, we measure the overhead of the controller's communication with LEGO Runtime by

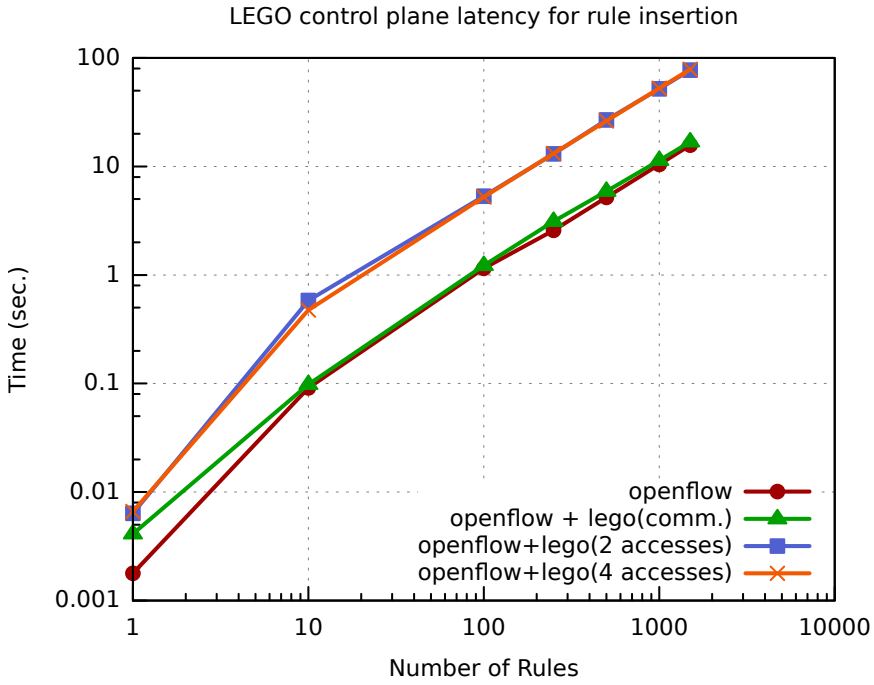


Figure 30: LEGO control plane rule installation time at 100 flows/sec. Increase in time for two and four register access, represents overhead of rule installation on custom packet processor

sending one blocking rule installation command on the LEGO Runtime and one blocking OpenFlow flow installation command as shown by "OpenFlow + LEGO(comm.)". This line shows there is no significant overhead while installing rule using LEGO controller.

On the other hand, the top two lines in Figure 30 show the latency to install one OpenFlow rule, plus making two and four register accesses on same CPP. We measured register accesses as it is the basic unit in order to perform any read or write on an FPGA based design. These two lines show LEGO's rule installation overhead, which is much higher than OpenFlow's rule installation. There are two reasons for this performance drop. First, LEGO's rule installation calls from the controller to LEGO Runtime are blocking. More importantly, our CPPs (*i.e.*, NetFPGA cards) use a PCI control interface, which is now over a decade old. These readings are comparable to FIB update latency reported in [136]. PCIe based CPPs will significantly reduce rule installation latency.

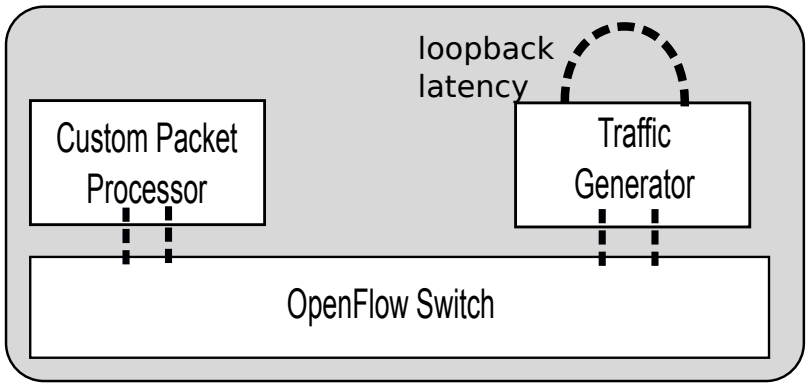


Figure 31: LEGO Data Plane Latency Setup. Loop back time is subtracted from the time it takes for packet to go through an OpenFlow switch and a CPP

5.4.2 Latency of LEGO Data Plane

Here we present LEGO's data plane latencies and compare them with the latencies of a hardware switch and other programmable data plane approaches.

Figure 31 illustrates our test setup for measuring data plane latency. We use two-foot Cat 6 Ethernet cables to minimize propagation delay. Cat 6 has a propagation delay of 5.5 ns per meter, which is negligible. To measure FIFO latency we start a nanosecond granularity clock and send two packets from the traffic generator: one through the loopback cable, and the other through the network. Once we receive these two packets back we note their arrival times; their difference constitutes the network latency.

5.4.2.1 Base Latencies

We first establish the baseline packet-forwarding latency of the OpenFlow switch that serves as the active switching backplane by sending a single packet to the switch and back to another port on the same NetFPGA. Figure 32 shows the latency overhead of the LEGO data plane with a single CPP. We show the mean of three latency measurements and repeat this process for different packet sizes. LEGO's minimal configuration only needs one CPP; so we compare the OpenFlow switch latencies with LEGO's latency with one CPP. For each of these measurements we manually

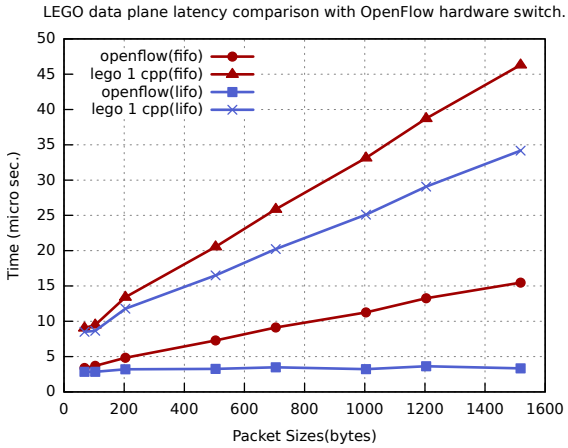


Figure 32: LEGO with one CPP compared with OpenFlow hardware switch.

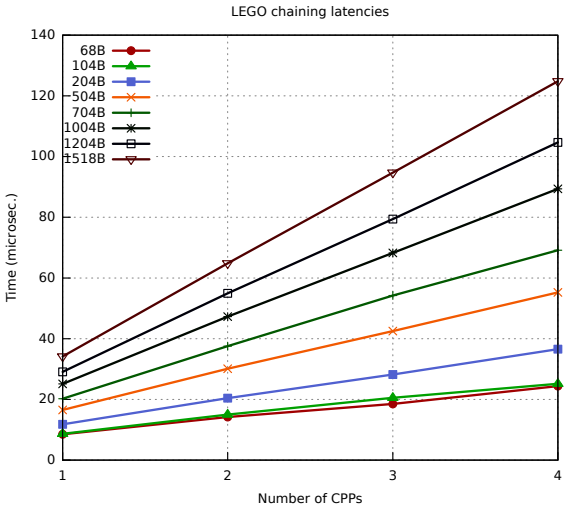


Figure 33: Effect of chaining on data plane latency.

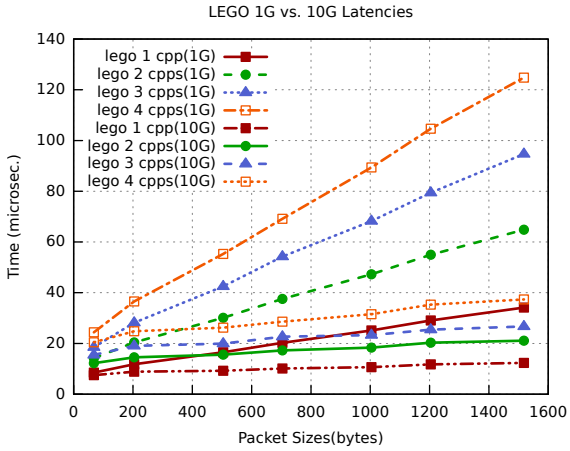


Figure 34: LEGO data plane 1G latencies compared with 10G latencies.

install for table entries in the OpenFlow switch.

RFC 1242 [34] recommends using LIFO for store-and-forward and FIFO approach of time stamping for cut-through switches. But to provide an apples to apples comparison [17], we report both LIFO and FIFO latencies, when comparing LEGO latency with OpenFlow switch. In other data-plane latency graphs measurements are based on LIFO latencies.

As can be seen in figure 32 even with a single programmable custom packet processor, LEGO latency is much higher than the OpenFlow switch used as active switching

backplane, for example for a packet of 68-bytes OpenFlow 1Gbps switch has measured latency of 3.4μs. but LEGO with one CPP has 9μs latency. These LEGO latencies are significantly worse when compared to OpenFlow hardware switch but are comparable with other commercial and academic forwarding planes. For example, [56] reports minimum latency of 24μs for 64-byte sized packets and 47.6μs for a packet requiring a single lookup over 10G Ethernet.

5.4.2.2 Chaining Latencies

Apart from having the ability to allow a single CPP for extra packet processing, LEGO allows more than one CPPs to be put together in a chain and perform processing on the incoming packets.

To measure latency of chains of different lengths we did the same experiment as mentioned in section 5.4.2.1, but this time we added more CryptoNIC-L CPPs in a packet's path. Figure 33 shows latencies for chains of size 1,2,3 and 4, for different packet sizes. As can be seen that with a chain of four FPGA based custom packet processors the latency is comparable to other programmable routers [56].

5.4.2.3 LEGO Technology

With 1G Ethernet technology LEGO's packet latency numbers are comparable to 10G-based programmable router latencies [56]. However, LEGO's chain-based designs requires packets to traverse different Ethernet interfaces frequently, which can waste time on packet serialization/deserialization with 1G technology—these times may reach 99.5μsecs for 1518-byte packets for a four chain configuration. (We note that 1G Ethernet technology has 10 times more serialization latency than the 10G Ethernet technology.) With many networks already moving towards 10G technology, we show LEGO packet latency with 10G serialization, as opposed to 1G serialization in Figure 34. With a chain of four CPPs the latency for packet passing through LEGO

switch is less than 21 μsecs. If we incorporate a software based packet forwarding device [56] and a hardware based CPP in a single chain we can still get good forwarding latencies. With availability of 40G and 100G Ethernet technology [1, 8, 31, 82] and 400G Ethernet under standardization [83], we think that serialization latencies will likely be negligible for LEGO's implementation using these technologies.

5.4.3 Packet Forwarding Rate

We use the setup shown in Figure 28 to evaluate LEGO's packet forwarding rate under different conditions.

In this section we explore various aspects of LEGO's data-plane throughput in more detail. First, we measure the overhead of adapting existing FPGA packet processors for LEGO's active backplane tagging protocol; our results show no measurable overhead. Second, we measure our setup's maximum throughput, demonstrating that we can process up to 34 Mpps when all 24 ports of the switch are flooded. Third, we show the feasibility of building chains of up to six custom packet processors.

5.4.3.1 Overhead of ASB Header Processing

We used the LEGO header to control packet processing because it could be implemented at a low level of hardware abstraction (*i.e.*, FPGAs). We measure the packet-forwarding overhead of introducing this additional header processing over existing FPGA packet processing elements. The setup for this experiment uses a single NetFPGA where all four ports are connected to another NetFPGA that is used for traffic generation using NetFPGA traffic generator [51]. We measure the forwarding performance of the two NetFPGA-based packet processing modules described in Section 3.4. Our evaluation shows that integrating the FPGA-based CPPs with LEGO incurs little additional overhead in comparison to the baseline performance of these modules.

We measure how integrating individual custom packet processors with the active

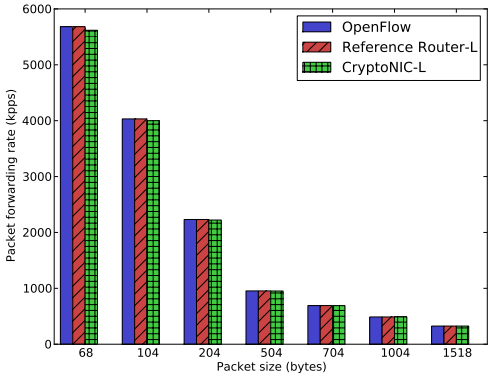


Figure 35: LEGO header based packet rates of individual designs vs. raw active switching backplane forwarding.

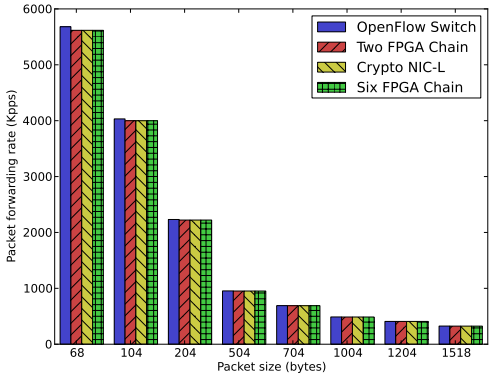


Figure 36: Packet Forwarding Rates for a Single Chain of Two FPGA cards and Single Chain of Six FPGA cards.

switching backplane affects packet forwarding rates for different CPPs. Figure 35 shows packet forwarding rates of the two hardware designs compares with the OpenFlow switch's packet forwarding speed without CPPs. The "OpenFlow" bar shows the packet-forwarding rates for different-sized LEGO header packets when we flood four ports of the OpenFlow switch and then receive all the packets on another NetFPGA card. This line establishes the baseline for single FPGA experiment in which traffic is being sent through active switching backplane.

Once we establish this performance baseline for packets with a LEGO header, we compare this baseline performance with Crypto-NIC-L and Reference Router-L designs. The "Crypto-NIC-L" bars show the packet forwarding rate when we install the Crypto-NIC-L design on the NetFPGA card that is doing the packet forwarding. The "Reference Router-L" bars show the packet forwarding rate when we download the IPv4 Reference Router-L design on the NetFPGA and the packet from source are sent to IPv4 Router-L design and then to the sink.

This experiment requires each packet to go through the switch twice: once from the source to the CPP and again from the CPP to the output port of ASB. We do not observe any significant drop in packet forwarding rates. In the case of Crypto-NIC-L, we observe a small drop in packet rate which we believe is not due to LEGO header

handling in CryptoNIC-L design, since the same design for LEGO header handling is being used by Reference Router-L.

5.4.3.2 Aggregate Backplane Throughput

We measure LEGO's throughput when CPPs and ASB operate at their full capacity. We use all six NetFPGA cards as packet forwarding data-planes and connect these cards to 24 ports of the switch in closed port configuration. We use six more cards as traffic generators, as shown in Figure 28.

This experiment has three purposes: (1) To prove feasibility of OpenFlow switch as active switching backplane for LEGO; (2) To ensure that LEGO's CPPHost(enclosing twelve CPPs), works correctly even when all twelve cards are used (six for packet forwarding and six for traffic generation); and (3) To determine the feasibility of a single PCI bus to support twelve cards, thus supporting up to twelve OpenFlow switches as LEGO ASBs.

To measure ASB throughput, we use six cards as forwarding devices and six cards as traffic generators and receivers. In this experiment active switching backplane is able to forward the packets without any packet drops with the configuration shown in figure 28. This experiment shows that the active switching backplane can forward traffic from all cards at full line rate. It also shows that LEGO can support up to six packet forwarding devices connected to single ASB; a single host can support up to twelve CPPs.

5.4.3.3 Chaining

To evaluate the feasibility of chaining, we connected two NetFPGAs to the active switching backplane and used a NetFPGA 1G traffic generator as a traffic source and sink. We pre-installed rules in the OpenFlow switch to send the traffic from source to first hop CPP, which was a CryptoNIC-L. We used this traffic from CryptoNIC-L and sent it to a Reference Router-L module to select the output port for the incoming

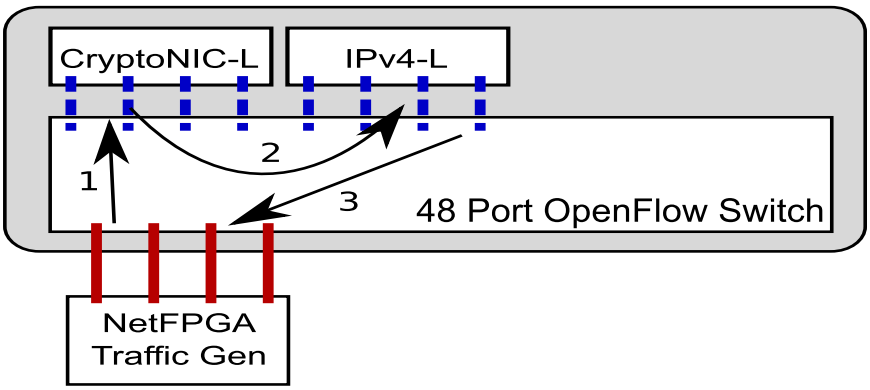


Figure 37: Experimental setup for a single chain of two custom packet processors, where packet stages are marked with numbers.

packets. Figure 37 shows the setup of this experiment. We use a two-stage chain that consists of two NetFPGA cards. The first CPP performs encryption or decryption on incoming traffic. The second CPP is the NetFPGA-based IPv4 reference router that provides the packet-forwarding capability to the traffic that is decrypted by the previous processing element.

Figure 36 shows the packet forwarding rates of the overall chain. Although the overall packet forwarding capability of two FPGA cards is 8 Gbps, after introducing a chain that consists of the two cards, the overall processing capability of two FPGAs drops by half. The throughput cost of chaining is expensive: apart from increasing the requirement for FPGA logic, chaining also increases the number of rules required to be installed on active switching backplane by at least a factor of two. In our simple scenario, where rules were based on the input and output port number, we assumed that all the traffic that goes to the first FPGA would also need to be forwarded to the second. The number of rules installed increased by a factor of 2.5, when compared with simple OpenFlow rule installation without any CPP.

The slight packet loss that appears in Figure 36 shows that the maximum forwarding capability of a chain is contingent upon the weakest link (in this case, the first CPP). In addition to verifying LEGO's chaining operations and measuring their forwarding rate, we wanted to see how many CPPs we could compose in a single chain.

We used six NetFPGA cards to create a single chain, with the six cards aligned on a single chain. Figure 36 shows that no packets are dropped, even when the packets go through a chain of six cards. We are certain that some drops occur due to the CryptoNIC-L module, but we see no additional drops due to chaining.

5.5 Summary

This chapter presents LEGO, an "inside-out" switch that lets network operators define a variety of custom packet-processing functions by expanding both the set of actions that the switch can perform and the set of conditions under which these actions might be taken.

LEGO's runtime abstracts away the complexity and diversity of CPPs and allows functions to be "stitched together" by installing flow-table entries using a standard SDN controller interface. LEGO allows network operators to compose custom packet processing hardware to build flexible and fast packet-processing pipelines that can provide an expanded set of actions to take on packets, as well as an expanded set of conditions compared to standard OpenFlow switches. LEGO co-opts the VLAN header to allow traffic flows to specify packet processing pipelines through one or more custom packet processors (hardware/software) that are connected by a programmable active switching backplane. Our evaluation shows that LEGO achieves latency and throughput that is comparable to state of the art OpenFlow switches, while permitting a much wider range of flow conditions and actions than existing switches.

CHAPTER 6

NETWORK FUNCTION PROGRAMMING ABSTRACTION

6.1 Introduction

Recent trends suggest that network operators seek to deploy an increasing range of network functions in the network. These functions can perform arbitrary functions on packets, including access control, intrusion detection, load balancing, caching, and transcoding. It is commonly—if not always—assumed that these functions should be deployed as monolithic middleboxes [60,69,88,90,137,156]. Until recently, these middleboxes have been deployed as vertically integrated hardware (*e.g.*, dedicated load balancers, firewalls, and other devices), although the shift towards network functions virtualization (NFV) [61] has enabled the deployment of these middleboxes in virtual machines [107].

Current approaches to NFV make it possible to place existing middleboxes in virtual machines at various points in the network and steer traffic through those middleboxes, instantiating and decommissioning instances in response to changing traffic conditions. This approach to deploy network functions imposes severe limitations. First, it requires the wholesale deployment of an existing middlebox; they do not allow an operator to implement custom, fine-grained packet processing functions in the data-plane that could be re-used across multiple applications. For example, many middleboxes may (re)implement their own packet processing modules that filter or

load-balance traffic, compute statistics on traffic flows, or otherwise perform operations on packets (*e.g.*, checksums) that could be shared across different functions. Second, deploying an entire middlebox inside a virtual machine does not scale to a large number of instances on any physical machine, and deploying (or migrating) the middlebox functions may be cumbersome in their own right.

We offer a fundamentally different approach to deploy network functions. Rather than the conventional approach of redirecting traffic flows through monolithic middleboxes, we propose a programming model that allows a programmer to specify which sequences of network functions should be applied to traffic that passes through the network, leaving the thornier questions of *where* in the network those functions are actually applied and *how* these functions are applied to the underlying runtime system.

This chapter presents *Slick*, an approach to programming network functions that allows an operator to implement network functions as chains of lightweight functions that can be placed at arbitrary locations in the network and composed into more complex packet processing sequences. Slick has two salient features:

- **Programming abstraction.** We develop a programming abstraction that allows a network operator to (1) implement custom network functions in a high-level language (*i.e.*, Python) and (2) specify which traffic flows should be routed through sequences of these functions. A programmer may implement (or reuse) specific functions as *elements* (a programming model that takes inspiration from Click [96]) and specify sequences of elements that should operate on specific portions of flowspace.
- **Runtime.** Slick's runtime scalably and efficiently implements the programming abstraction we have designed by decomposing network-wide packet processing into constituent functions and placing those functions at appropriate locations in the network. In contrast to existing approaches, which consider placement

in the absence of steering [98, 133, 137], or vice versa [68, 71, 90], Slick takes a holistic approach, performing *both* placement of modular packet-processing elements and steering of traffic through those elements.

In contrast to NFV—which concerns the instantiation and management of existing monolithic middleboxes in virtual machines—Slick allows the placement of fine-grained functions, specified as *elements* that the programmer can write in a high-level programming language (*e.g.*, Python), making the placement of these functions more nimble, taking better advantage of available network resources, and allowing potential reuse and sharing of network functions that are applied to traffic. Slick determines how many instances of each element should be instantiated and where individual elements should be placed ("placement"), as well as which traffic flows to direct through specific element instances ("steering"). Slick elements can be reconfigured at runtime after they are installed, and Slick policies can specify that placement or steering should change at runtime, in response to triggers from the network. For example, a middlebox that checks DNS requests against a blacklist could trigger all of the user's traffic to be steered through the closest deep-packet inspection element.

We develop several placement and steering algorithms and evaluate them on enterprise and data-center network topologies. Our evaluation shows that Slick's heuristics can achieve near-optimal network bandwidth utilization on many network topologies and can reduce the average link utilization compared to an approach that only uses consolidation by as much as a factor of two.

The rest of the chapter is organized as follows. Section 6.2 provides and overview of the Slick architecture and its use case in real network. In section 6.3 we describe the Slick programming abstraction and how it can be used to write network functions and then program them inside the network. We finish this chapter with summary of the Slick programming abstraction in section 6.5.

6.2 Slick

In this section, we present an overview of Slick, describe a motivating example (and explain why this example is difficult to implement in existing NFV architectures), and describe Slick's programming model and runtime.

6.2.1 Overview

Figure 38 illustrates Slick's architecture. The Slick controller runs an application that specifies a sequence of elements that should process a particular portion of flow space. An *application* specifies which traffic should flow through specific sequences of *elements*. The Slick *controller* supports these applications by deploying elements (on top of a *shim* on each machine) and installing forwarding rules in the switches to direct traffic through particular sequences of elements. The controller instantiates functions on *machines* and installs forwarding rules in *switches* to steer traffic towards those machines. The Slick runtime takes a high-level policy and determines the number of element instances to deploy (and where to deploy them) to ensure that no single element or network link is overloaded and that traffic sees good end-to-end performance. Given values for each packet-header field, the controller determines the sequence of elements that should be applied to a particular flow and installs flow table modifications into corresponding switches to ensure that the respective flow is forwarded through the corresponding sequence of element descriptors.

Motivating Example. Suppose an operator configures the network so that all Web traffic traverses an intrusion detection system (IDS) [120, 142]. The application specifies that all Web traffic (*i.e.*, TCP traffic with port 80) flows through an IDS element, with all packets of a TCP connection in both directions traversing the same element. The controller deploys one or more IDS elements in the network and installs rules in the switches to direct port-80 traffic through the element. As traffic demand

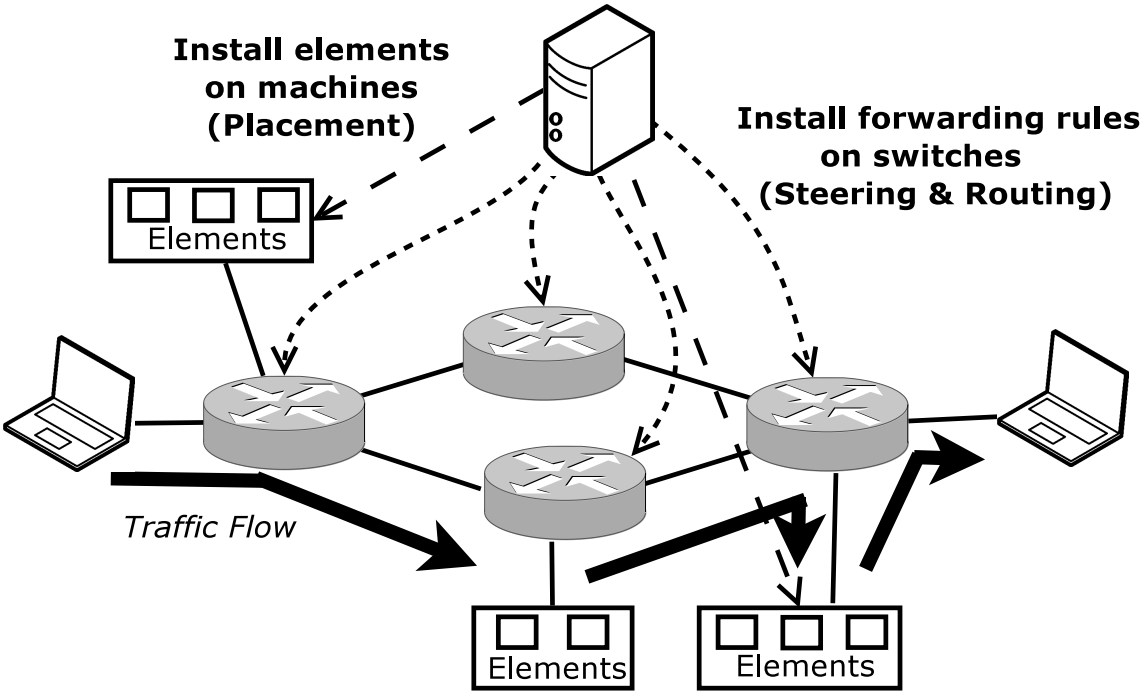


Figure 38: Slick architecture. A programmer writes a Slick program that runs at the controller, which in turn installs elements (*i.e.*, high-level functions) on machines in the network (*placement*) and installs forwarding rules on switches to direct traffic flows through sequences of elements (*steering*).

increases, any single IDS element may become overloaded; at this point, the Slick controller instantiates a second IDS element and splits the port-80 traffic over two IDS elements, taking care to ensure that ongoing TCP connections complete at the original IDS element and only new flows traverse the second element. If the traffic demand decreases to the levels from before the controller added additional elements and all flows through the first IDS element expire, the controller reclaims resources by removing that IDS element instance. Over time, the controller monitors the machine and network load, adjusting the traffic splitting and routing to minimize congestion. The IDS itself might inspect network traffic and perform deep-packet inspection (DPI) only when it observes DNS traffic from a device on the network to a blacklisted DNS domain.

6.3 Programming Abstractions

Each network function corresponds to a software *element*. An element may be configured either at initialization time or dynamically; it may also generate an event stream that sends events to the controller. A Slick control *application* specifies a high-level policy, indicating which traffic flows should traverse a particular sequence of elements (*e.g.*, packets with destination port 80 should traverse a firewall followed by a transcoder); an operator can write such a policy independently of the network topology or where the elements are installed.

Slick supports modular, composable elements that permit reuse across many applications; each element also supports dynamic configuration and supports sending events to the Slick controller that might subsequently affect its operation. Slick elements are inspired by elements in Click modular routers [96], from which we derive Slick's name. In this section, we describe how to program functions and applications, detailing the interfaces they expose and the abstractions presented to them.

6.3.1 Writing Slick Elements

Slick elements run on machines; an element can be an arbitrary executable and may also have state. Elements process packets, handle configuration requests from applications, and send events to the controller.

Element methods. When a controller first installs an element on a machine, it invokes the element's `init()` method. As packets destined for that element arrive at the machine, the element's `process_pkt()` method is called; this method can perform arbitrary packet processing. An element can also be configured dynamically by the controller: the `configure()` method allows the controller to dynamically reconfigure network elements. This method also allows a controller to update an element's internal state; for example, a firewall element could accept new rules via `configure()`.

```
1  class Logger(Element):
2     def __init__(self, shim, ed):
3        Element.__init__(self, shim, ed )
4        self.file_handle = None
5
6     def init(self, params):
7        filename = params["file_name"]
8        filename += str(self.ed)
9        if(filename):
10          self.file_handle = open(filename, 'a+', 0)
11
12    def process_pkt(self, packets):
13       for buf in packets:
14          flow = self.extract_flow(buf)
15          self.file_handle.write(str(flow) + '\n')
16       return packets
```

Figure 39: The `Logger` element logs all packets it receives. (We have elided the element's `shutdown` method for clarity.)

Finally, an element can issue asynchronous, distributed *triggers* that allows it to send events to the controller. The `raise_trigger()` method accepts arbitrary inputs and delivers them to the controller (who, as we will see, delivers them to the proper applications' trigger handlers). Figure 39 shows an example of a simple Slick element that logs all packets that it sees. The `init()` method (lines 6–10) performs any operations that should be called when the element is initialized (in this case, opening a file); the `process_pkt()` method (lines 12–16) is invoked whenever the element sees a packet.

Two properties of the `Element` class design make it easy to reuse elements across applications. First, elements need not specify the traffic flows that they process; an element simply processes *any* flow that is passed to it. Second, elements are agnostic about what application is invoking them. For example, the `TriggerAll` element sends an event to the controller, and *any* control application that registers for these events will receive them. Because any element implementation is agnostic about both the subset of traffic that it will operate on and the applications that will instantiate it, any given element implementation can be reused across a wide variety of applications.

6.3.2 Programming Slick Applications

Slick applications run at the controller. These control applications specify a sequence of elements that should process a given portion of flow space (*e.g.*, send all port 53 traffic through a `Logger` element).

Instantiating elements. An application specifies a portion of flow space and applies that flow to a particular element/elements using the `apply_elem()` method. Applying an element to a portion of flow space causes the controller to install that element at the appropriate locations in the network.

Figure 40a shows an example `HttpLogger` control application. Lines 7–9 specify that the controller should ensure that `Logger` (Figure 39) operates on all traffic with destination port 80, and to supply `http.log` as its input parameter (which will set the log's filename). The `apply_elem()` method (Line 10) takes as inputs the flow to which an element should be applied, the name of the element, and an optional set of parameters to send to those elements' `init()` method. Each call to `apply_elem()` creates a new instance of the specified elements.

A Slick application may create multiple instances of multiple elements. For example, the `HttpLogger` application could have made another call to `apply_elem()` on all port 443 traffic with another `Logger` function to also log HTTPS traffic. The `apply_elem()` method returns a unique element descriptor for each instantiated element, to allow the controller to configure these elements after installation time, and to process triggers.

Interacting with elements. An application can also interact with any installed element after the element has been installed in the network. Applications use `configure()` with the corresponding element descriptor to send arbitrary configuration messages to Slick controller, which will ultimately result in a call to that element

```
class HttpLogger(Application):
   def __init__(self, controller, ad):
      Application.__init__(self, controller, ad)

   def init(self):
      parameters = [{"file_name":"/tmp/http_log_mach"}]
      flow = self.make_wildcard_flow()
      flow['tp_dst'] = 80
      flow['nw_proto'] = 6
      ed = self.apply_elem(flow, ["Logger"], parameters)
      if(self.check_elems_installed(ed)):
         self.installed = True
```

(a) Logging all port-80 traffic at in-network traffic elements.

```
class HttpLoggerViaTrigger(Application):
   def __init__(self, controller, ad):
      Application.__init__(self, controller, ad)

   def init(self):
      flow = self.make_wildcard_flow()
      flow['tp_dst'] = 80
      flow['nw_proto'] = 6
      self.ed = self.apply_elem(flow, ["TriggerAll"])
      if(self.check_elems_installed(self.ed)):
         self.installed = True
         self.file_handle=open("http.log", 'a')

   def handle_trigger(self, ed, msg):
      if(ed in self.ed):
         self.file_handle.write(str(msg))
```

(b) Logging all port 80 traffic at the controller.

Figure 40: Two implementations of `HttpLogger` that perform logging in different locations.

instance's `configure()`. When an element sends a trigger to the controller, the controller calls the corresponding application's `handle_trigger()` method and passes it two values:the descriptor of the element that raised the trigger and any associated data. `HttpLoggerViaTrigger` in Figure 40b applies the `TriggerAll` element (which simply raises a trigger for every packet) to all HTTP traffic; `handle_trigger()` will

thus be called with each HTTP packet sent in the network.

Choosing where functions are performed. Figures 40a and 40b illustrate how Slick's programming model allows a programmer to chose where processing takes place: (1) `HttpLogger` places all the work in the in-network machine by having the `Logger` element capture and log all of the packets to file; (2) `HttpLoggerViaTrigger` uses the `TriggerAll` element to cause the controller to log all packets at the controller application. These implementations represent two *extreme* design points. The first approach places all processing at the elements themselves, which is similar to how middleboxes operate today. This approach scales well, depending on where elements are installed in the network. The latter approach places all processing at the controller, which can introduce a bottleneck at the controller.

Building applications from multiple elements. Slick applications can also define interactions between multiple elements. Figure 41 shows a `BlacklistDropper` application, which also illustrates the use of `raise_trigger()` and `configure()` in the `DNSBlacklist` element. The application applies `DNSBlacklist` element to all outgoing DNS traffic (line 5), which raises a trigger whenever it detects a DNS lookup to a blacklisted domain (lines 23–28). When the application receives this trigger, it installs the `DropAll` element that simply drops all packets (lines 11–14), applying it to all subsequent traffic from the host that initiated the DNS lookup (line 14).

Slick also allows element chains, enabling sequential processing of packet flows by the elements in the chain. For example, to log all the port 80 traffic and subsequently drop all the traffic, we can modify (line 10) in Figure 40a as follows:

```
eds = self.apply_elem(flow, ["Logger", "DropAll"], parameters)
```

```
class BlacklistDropper(Application):
    def init(self, blacklist):
        flow = self.make_wildcard_flow()
        flow['tp_dst'] = 53
        eds = self.apply_elem(flow, ["DnsDpi"])
        if(self.check_elems_installed(eds)):
            self.installed = True
        droppers = list()

    def handle_trigger(self, ed, trigger):
        if(trigger['type'] == 'BlacklistedQuery'):
           src_flow = self.make_wildcard_flow()
           src_flow['nw_src'] = trigger['src_ip']
           eds = apply_elem(src_flow, ["DropAll"])
           if(self.check_elems_installed(eds)):
               droppers.append(eds[0])


class DNSBlacklist(Element):
    def init(self, blacklist):
        self.blacklist = blacklist

    def process_pkt(self, pkts):
        domain, src_ip = extract_dns_domain(pkts)
        if(domain in self.blacklist):
            self.raise_trigger(self.ed,
                {'type'   : 'BlacklistedQuery',
                 'src_ip' : src_ip })
        return pkts

    def configure(self, params):
        if(params['command'] == 'set_blacklist'):
            self.blacklist = params['blacklist']
```

Figure 41: Slick applications can use triggers to asynchronously compose elements. Element descriptors disambiguate multiple instances of the same element.

6.4 Implementation

We have implemented this programming abstraction using Python programming language. Slick applications, elements, shim layer and communication mechanisms are implemented using same language but we believe that the programming abstraction is

general enough that it can be implemented in any other programming language such as C, C++, Java etc. Here we note that the applications and elements don't need to be written in same programming language and they can be written in different programming languages. In following sections we provide further details:

6.4.1 Elements and Applications

To demonstrate the flexibility and generality of Slick's programming model, we have implemented nearly 15 elements, which we have incorporated into several real-world applications. The Slick elements provide functions at different granularities and levels of complexity. These function include network traffic logging, TCP Stream analysis to detect OS and browsers, DNS deep packet inspection, encryption, decryption, compression, and decompression. The applications we have implemented include a traffic quarantine application that is triggered by DNS-based traffic monitoring and an application firewall.

6.4.2 Shim.

The Slick shim layer makes it possible to deploy and decommission elements at run-time and also includes a virtual switch to multiplex and de-multiplex traffic through these elements. The shim also allows Slick to marshal control messages between Slick's control applications running on the controller and the elements. Control messages and triggers between applications and the controller are encapsulated in JSON and sent over TCP connections.

6.5 Summary

In this chapter we have presented an abstraction that can be used to program network functions without worrying about their deployment. There are two main parts of this programming abstraction, Slick element abstraction and Slick application abstraction. Slick element abstraction allows programming of network functions without worrying

about the context where they'll be deployed. Slick's element programming abstraction can be used to add data plane functions inside the network. The rationale behind Slick's element programming abstraction is to enable the network function writer to focus more on the functionality that he/she wants to add inside the network instead how the function will interact with underlying physical hardware.

Once the element code is written, Slick's application programming abstraction can be used to deploy the function code inside the data plane. In next chapter we discuss how the programming abstraction can be supported by a runtime that employs different algorithms to deploy new functions/enhancements to network data plane. Then we evaluate this runtime with the programming abstractions developed in this chapter.

CHAPTER 7

NETWORK FUNCTION DEPLOYMENT

7.1 Introduction

The Slick *runtime* using Slick *controller* maps a control application's high-level policy to the available pool of network resources (*i.e.*, available network bandwidth and computational elements). Given a high-level policy, the controller determines how many instances of each element to deploy and where to place or migrate them (*placement*). The controller also determines the paths that each traffic flow should take through the network so that traffic flows are processed by the correct sequence of elements and also experience good end-to-end performance (*steering*). The controller must adapt to topology changes and machine failures, as well as shifts in load and changes in the high-level policy. A *shim* on each machine allows the controller to interact with the elements (*e.g.*, to configure the element and receive triggers, shown in § 6.3).

The Slick controller maps each new flow to elements that are installed on machines in the network and keeps an updated view of what resources are available on each machine. Instead of performing a single optimization given resources and traffic flows, the Slick controller performs a continuous *incremental optimization* that minimizes changes to the installed configuration and ongoing network traffic flows.

Rest of this chapter is organized as follows. In section 7.2 we show how network functions can be deployed inside the network using different placement algorithms and present multiple heuristics that can be used to minimize the overall network resource utilization while placing network functions. In section 7.3 we present an overlay network abstraction that can be used to steer traffic through the data plane

Table 11: How different placement heuristics help achieve Slick objectives.

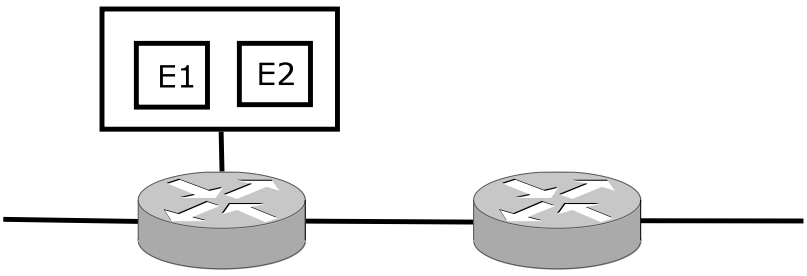
Heuristic	Reduce b/w utilization	Reduce resource utilization	Reduce Latency
Consolidation	Yes	Yes	Yes
Inflation	Yes	Yes	

functions/enhancements already placed inside the network. Section 7.4 briefly describes how end to end connectivity is provided in Slick runtime. In section 7.5 we present the implementation of Slick runtime that uses Slick *placement*,*steering* and *routing* modules to deploy network functions. In section 7.6 we then present evaluation of Slick's placement and steering algorithms and its controller's performance on a variety of network topologies. We finish this chapter with its summary in section 7.7.

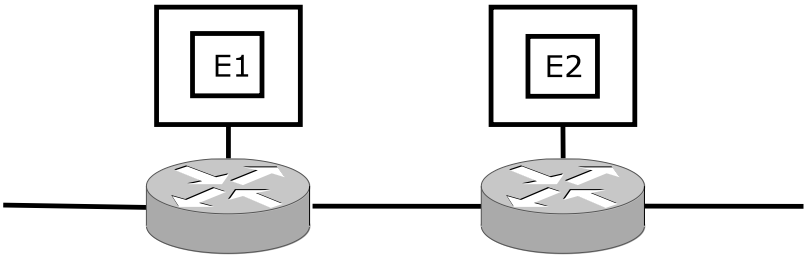
7.2 Placement

The controller's placement algorithm determines the machines in the network where element instances should be installed. The placement algorithm may ultimately place multiple instances of the same element at different places in the network, and a single machine may also host multiple elements.

Placement aims to place instances of elements at various machines in the network to ensure that flows are processed by their corresponding element sequences while using a reasonable amount of bandwidth and machine resources and ensuring a low-latency end-to-end path. Slick uses an *inflation heuristic* to reduce the overall network bandwidth required to support element sequences and a *consolidation heuristic* to reduce both the utilization on individual links and the number of overall machines required to host element instances. Table 11 summarizes how different heuristics help achieve different placement goals or have no impact on Slick's goals. Placement applies these two heuristics in order: the controller first decides whether (and how) to consolidate elements on physical machines; second, the controller determines where to place the consolidated elements.



(a) Consolidation.



(b) No Consolidation.

Figure 42: When making placement decisions, the Slick controller must determine whether to consolidate multiple elements on a single machine or distribute those elements across multiple machines in the network or use a combination of the two.

Step 1: Consolidating elements. When we have more than one element that should operate sequentially on a certain flow space, the first step is to decide whether we should consolidate contiguous elements onto a single machine, or if we should distribute them across multiple machines. Consider the network in Figure 42, which shows two possible configurations in which a chain of two elements can be deployed.

We define an *inflation factor* as $\log(f_{out}/f_{in})$, where f_{in} and f_{out} are the input and output traffic volumes, respectively. The intuition for consolidation is that elements with negative inflation factor should be placed closer to sources, and elements with positive inflation factor should be placed closer to destinations. For any ordered list of elements $(E_1, \ldots, E_n)$, we can decide places to "break" the list into any number of sub-lists, where each sub-list is placed on a single machine.

We can define the inflation factor of a machine m, λ_m, as the sum of all of the inflation factors of the respective elements placed on that machine. A negative inflation factor thus means that the consolidated elements on that node decrease overall traffic,

Table 12: The inflation heuristic helps determine whether an element should be placed closer to the source or closer to the destination.

Element	Placement
Negative Inflation	
Access Control	Source
Firewall	Source
Intrusion Prevention	Source
Deduplicator	Source
Decapsulator	Source
Compressor	Source
Positive Inflation	
Encapsulator	Destination
Decompressor	Destination
Zero Inflation	
Deep Packet Inspection	Any
Intrusion Detection	Any
Stateful load balance	Any
Stateless load balance	Any
Encrypt	Any
Decrypt	Any

and vice versa for a positive inflation factor. Then, for a path of length l, we can define the inflation for some consolidation along that path p, λ_p as $\sum_{i=1}^{l}(i - l/2) \cdot \lambda_i$. The brute-force consolidation algorithm searches all possible consolidation combinations to minimize total inflation. Given M possible machines on which to place a sequence of E elements, the algorithm tests $\sum_{i=1}^{E} \binom{E-1}{i-1} \cdot \binom{M}{i}$ possible combinations.

Table 12 enumerates some example elements, their inflation factors, and whether they should be placed closer to source or destination. The priority of placing an element near sources or destinations can be overridden by Slick application writer.

Step 2: Placing consolidated elements. Once minimum-cost consolidation is computed, the placement algorithm uses the flow connectivity matrix for each flow space, where c_{ij} is the number of flows from i to j. The placement algorithm identifies the longest common routing path between the source(s) and destination(s). It then

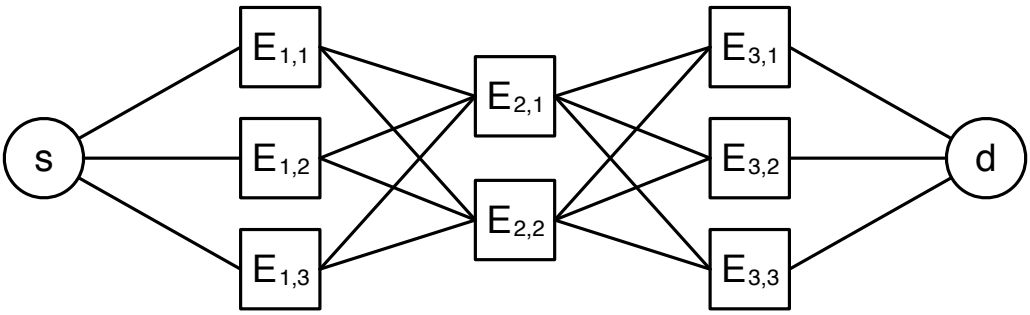


Figure 43: Slick uses a virtual topology with m_i elements at each stage i to decide how to steer traffic from source to destination in the order specified in the Slick application.

places consolidated elements with negative inflation factor on the node of longest common routing path that is closest to source(s), for elements with positive inflation factor, the algorithm places the consolidated element on the node of the longest common routing path that is closest to destination(s).

Elements with inflation factors near zero should be placed at machines that minimize the average path length for all source-destination pairs in the flow space, or that have the highest betweenness centrality for all source-destination pairs that exchange traffic in a given flow space. The betweenness centrality [100] of a vertex v, c_v is given by the expression:

$$c_v = \sum_{v \notin \{s,d\}} \frac{\rho_{sd}(v)}{\rho_{sd}} \tag{1}$$

where ρ_{sd} is the total number of shortest paths between s and d and $\rho_{sd}(v)$ is the number of those paths that pass through vertex v.

7.3 Steering

Given elements placed in the network and a flow that must traverse a sequence of elements, the steering module determines the specific sequence of element instances that a given flow should pass through. If there are multiple instances of a particular element, the steering module determines which element instance should be used to send traffic through a particular sequence of elements. The steering module acts on

a virtual topology that includes the elements and the connectivity between them.

Steering determines, for each portion of flow space, the specific sequence of element instances that should be used to process traffic for that flow. Recall that any given element might be installed in more than one place in the network; steering thus determines the instances of each element that traffic for a particular flow space should be routed through. Slick performs steering by constructing a virtual topology that represents the sources, destinations, and possible sequences of element instances at each stage of an element sequence; given this virtual topology, it computes a lowest cost path through the corresponding sequence of elements, for each portion of flow space. We describe this process in more detail below.

A Slick program determines the sequence of elements for each corresponding part of flow space; each element may have multiple instances in the network. Given an element sequence $\{E_1, \ldots, E_n\}$ for some portion of flow space, where any E_i may have multiple instances, Slick must be able to steer each traffic flow through any instance of each element in the sequence.

To help Slick compute the appropriate sequence of element instances for each portion of flow space, we represent the set of all element instances as a virtual topology, as shown in Figure 43. Traffic from s to d is routed through one instance of E_i, in order, from E_i to E_n. Each edge in the virtual topology has a weight that corresponds to the sum of the physical network distance multiplied by the anti-log of the inflation factor. This gives us weight of each virtual edge based on the physical network topology and inflation factor of the element instances. For a flow to which n elements are to be applied, this graph takes $O(n + \prod_{i=1}^{n} m_i)$ time to construct, where m_i is the number of element instances at stage i. Given this virtual topology, Slick computes the shortest weighted path from s to d.

To avoid overloading specific element instances, Slick removes machines from the

virtual topology if their load exceeds some operator-specified threshold. If no machines that host instances of some element E_i have spare capacity (again, determined by an operator-specified threshold), the Slick controller will provision another instance of the element on a new machine with the help of the placement module.

7.4 Routing

Given a specific sequence of element instances to forward traffic through, the routing module installs flow table entries into switches to ensure that a traffic flow follows a specific path between each pair of installed elements in an element sequence. It enables the steering module to implement *asymmetric steering* such that ingress and egress paths of the same flow can be asymmetric [134]. It also provides Slick runtime with network link information and placement module about the active switches generating traffic for a given flow space. Slick's routing module simply implements shortest-path routing between two element instances, although the module itself provides for other possible routing decisions between pairs of elements.

7.5 Implementation

We implemented Slick in about 15,000 lines of Python, with Slick's controller built on top of POX [124] controller as an SDN application. About half of the code involves the basic controller functions, such as communication with elements and interfacing to placement and steering modules, as well as the element shim as shown in Figure 44. The remainder of the code includes several elements and reference applications that use them.

The controller implementation includes functions to discover topology and machine resources, as well as the runtime that implements placement, steering and routing.

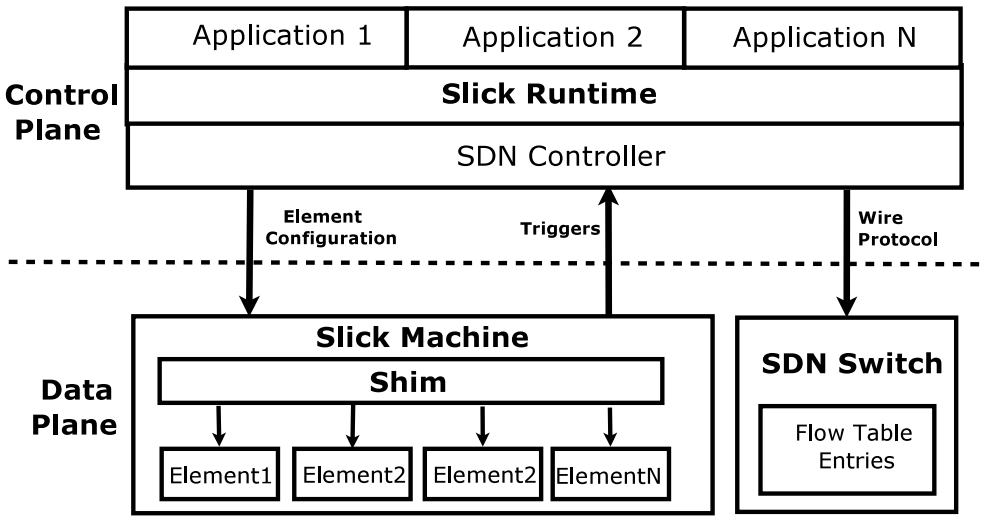


Figure 44: The Slick runtime operates on top of an existing SDN controller (in our implementation, POX), and hosts applications that specify functions that should operate on different parts of flow space. The controller installs and configures elements on machines in the network, which interface to the controller via a shim (*Placement*). The controller also uses a wire protocol (*e.g.*, OpenFlow) to configure flow-table entries in switches to steer traffic through the appropriate elements installed on machines (*Steering*).

7.5.1 Discovery

The Slick controller must discover both the network topology, the machines in the network that can host packet processing elements, and the current network conditions (*e.g.*, available network resources, current machine load). It discovers topology using a link-layer discovery protocol (*e.g.*, LLDP) and machine resources through a custom resource discovery protocol.

Network topology and congestion. Network switches and servers are discovered using OpenFlow's link-layer discovery protocol (LLDP). The controller maintains a network map that includes a mapping of element instances (each of which is identified by an element descriptor) to its location in the network topology, as well as a mapping between the MAC addresses that the controller knows about and their corresponding IP addresses. The controller also periodically polls the traffic load of each network link and the amount of traffic that each element is processing.

Machine resources. Each Slick machine runs a shim layer that registers with the Slick controller; the controller keeps track of the available machines on which it can install elements. Each machine's shim has a configuration file that contains information about that machine's available resources and any other constraints that exist; this specification ensures that the controller only installs elements on machines that have both the capability and the available resources to perform the corresponding processing (*e.g.*, a configuration might ensure that a certain encryption element is only installed on machines with the corresponding hardware acceleration for cryptographic operations). These specifications also include various other parameters including the number, types, and speeds of the processors on the machine, available storage, and the operating system type and version of the machine.

Network model and overlay network abstraction. Using information about available machines and link loads, the controller builds a network model to perform operations including (1) finding machines that can host a particular element (for placement); (2) finding machines where specific elements have been installed; (3) avoid routing new traffic flows through either congested links or loaded elements. Ultimately, the controller uses these functions to construct an overlay network for each network policy that includes the elements that are pertinent to any particular flow space.

Using knowledge of the underlying network topology and machine resources, Slick places elements and maintains an overlay network topology that abstracts the physical topology. Each policy has a corresponding overlay network topology; the steering module uses this overlay network to find, for each flow, the shortest path between the source and destination that traverses a particular sequence of elements.

7.5.2 Runtime

We implemented a variety of placement, steering, and routing algorithms in Slick. Slick implements several placement algorithms, including placing elements on k random machines in the network, placing nodes according to centrality, placing elements on compatible machines in a round-robin fashion, and weighting placement according to centrality on a graph with nodes weighted according to traffic load. Each placement algorithm is several hundred lines of Python. Slick implements steering according to random paths through the virtual topology (Figure 43), shortest hop count through the topology, and two different shortest paths through the virtual topology: one based purely on link weights, and another where link weights are assigned according to traffic loads. Each steering algorithm is between about 50 and 200 lines of Python. Routing is based on shortest paths in the underlying topology through the sequence of elements that steering selects; for this function, we were mainly able to rely on path setup functions in POX, but we also implemented a mechanism to route on shortest paths through the underlying topology. Slick's routing algorithm generates microflow forwarding table entries, which creates the potential for a large number of flow-table entries. Other work has explored ways to reduce the number of flow tables installed in switches, and Slick may be able to exploit these techniques [63, 125].

7.6 Evaluation

We evaluate Slick using Mininet [80] emulations for a variety of traffic matrices and topologies. We address the following questions: (1) What is the performance of Slick's placement and steering algorithms? (2) How efficiently does Slick place network elements and steer traffic through these elements? (3) How close are Slick's placement and steering algorithms to the optimal solution? (4) How does Slick generalize across different types of network topologies?

7.6.1 Experiment Setup

We evaluate Slick using the Mininet network emulator [80]. We opted for evaluating Slick using emulation rather than simulations or testbeds because emulations allow us to evaluate Slick under a variety of network topologies and with a variety of traffic matrices while ensuring that our results faithfully replicate the dynamics of real networks. We ran the Slick controller on one virtual machine and performed network emulation using Mininet on another. Each virtual machine had eight cores assigned and both VMs were running on a server with 16 cores(Intel Xeon E5620 @ 2.40GHz) and 24 GB RAM. The Mininet emulator limits our evaluations to topologies with less than 60 switches. For all evaluations, we use the applications and elements discussed in Section 4.4.

Topology. To demonstrate Slick's generality, we emulate a number of network topologies representing data-centers and enterprise networks. We evaluate Slick using a Fat Tree [18] network and using a canonical tree topology that is representative of small data centers [27] and enterprise networks [92]. In each topology, we assume that a Slick machine is attached to all switches within the network, so each machine that is attached to a switch can also host Slick elements. For all the experiments we use fat-tree with 20 (K=4) switches and tree topologies with 3 tiers and 15 nodes, except where stated otherwise.

Traffic Matrices. We evaluate Slick using a combination of two types of traffic matrices. (1) *East-West* traffic, emulating machine-to-machine traffic patterns which are prevalent in modern data-centers [27], (*e.g.*, MapReduce workloads). This traffic matrix generates traffic solely between end hosts within the same network; and (2) *North-South* traffic, emulating user-to-server traffic patterns that exist in a number of networks including data centers, enterprise campus networks, and WAN. Traffic

is between servers at the edge and the core-devices which act as a gateway to the Internet.

Evaluation Metrics. We evaluate Slick's effects on data-plane resource utilization and the performance of the Slick controller. First, we study the effects of Slick's programming model and algorithms on the overall network data plane utilization (Section 7.6.2). We show how using Slick's programming model and algorithms can help efficient implementation of Slick policies. To evaluate the efficiency of Slick's implementation of network policies, we focus on the following metrics: the sum of the average link utilizations (aggregate average network bandwidth), which allows us to understand the efficiency of the different algorithms; path length, which allows us to understand the impact of the different algorithms on the performance of individual flows; and link utilization, which also allows us to understand the effects of different algorithms on network traffic aggregates. In Section 7.6.3, we study the effects of network size, the length of Slick element chains, and the number of Slick element instances on the performance of the Slick controller.

7.6.2 Efficiency

We now evaluate the outcomes of the placement and steering that Slick computes. In doing so, we focus on evaluating Slick's performance against an Optimal algorithm, which provides an upper bound on Slick's performance; and a Random algorithm, which provides a reasonable lower bound on Slick's performance. The Random placement algorithm randomly places elements and Random steering algorithm randomly chooses which traffic to steer through which elements, while the Optimal algorithm assumes that all elements are placed at all locations and that each node has infinite capacity, thus eliminating the need for placement and steering. The Optimal algorithm ensures that the shortest paths are used at the cost of employing more elements.

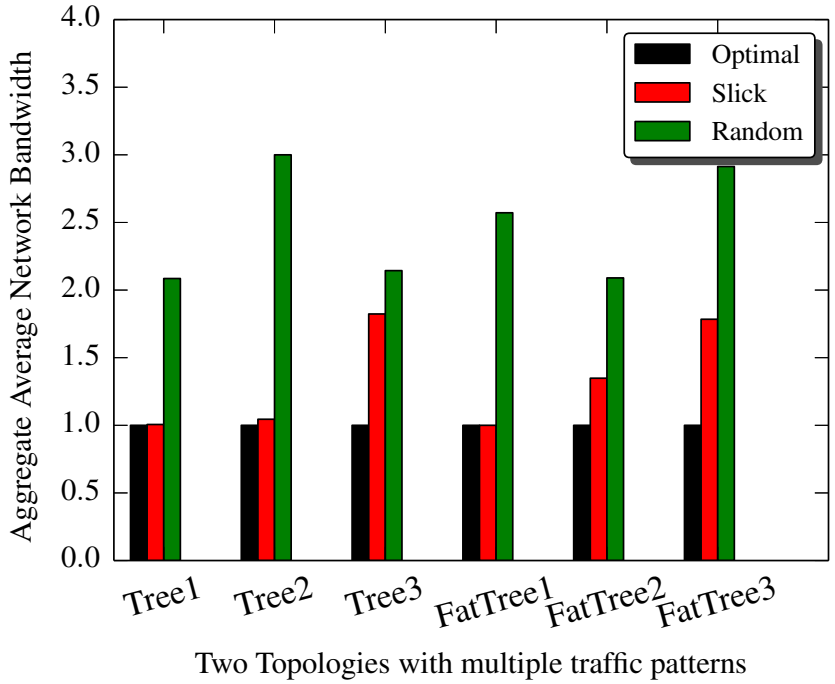
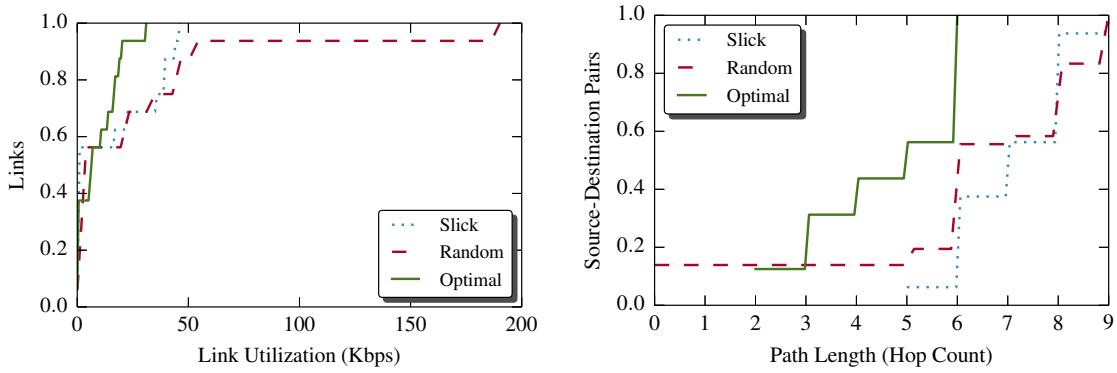


Figure 45: Network utilization under different algorithms: Slick, Random, and Optimal.

We have evaluated Slick, Random, and Optimal algorithms under all topologies and traffic matrices. Due to space constraints, we focus on the results from the largest emulations, but the results from smaller experiments are qualitatively similar. In Figure 45, we present the total bandwidth utilization from running the three algorithms on the tree topology and fat-tree topologies. The *Tree1* and *Fat-tree-1* experiments make decisions on four different flow spaces, which have both East-West and North-South traffic flows. We deploy four element chains with one to two elements in each chain. For each flow space, all of the sources are clustered in single rack and all of the destinations of a flow space are in single switch rack. In the *Tree2*, *Tree3*, *Fat-Tree-2*, and *Fat-tree-3* setups, sources and destinations are randomly distributed across the network. *Tree2* and *Fat-tree-2* have eight randomly selected source destination pairs and *Tree3* and *Fat-Tree-3* have sixteen randomly selected source destination pairs.



(a) Link Utilization

(b) Path Lengths

Figure 46: Comparison of Slick placement with Random and Optimal placement algorithms.

Slick consistently outperforms Random varying between 20% and 120%. Interestingly, for the traffic matrices where sources are clustered in single rack as well as destinations, (tree-1, fat-tree1 in Figure 45), Slick performs within 5% of Optimal. For topologies where sources and destinations are randomly distributed across the network, Slick performs between Random and Optimal: consistently reducing the performance gap between Optimal and Random by half.

We also examine the link utilizations, path lengths, and number of element instances in the resulting solutions (Figure 46a and Figure 46b). Although Slick has comparable path lengths (Figure 46b) and number of elements as Random (Slick and Random use one element instance and Optimal uses 20 element instances for the Fat-Tree topology), Slick achieves much lower link utilization than Random for the same number of element instances. The link utilizations that Slick achieves are comparable to those achieved by Optimal. Moreover, Optimal can only maintain shorter paths at the cost of deploying significantly more elements: In this experiment, Optimal uses N times more elements than Slick, where N is the number of switches in the topology.

Comparison to CoMB's "Strict" Consolidation. Slick uses inflation rates to guide consolidation and placement. CoMB [133]) also utilized consolidation as a way

to reduce overall network utilization. CoMB argues for a strict consolidation, which *always* consolidates all elements in a chain onto one machine. We examine network utilization under CoMB and Slick's consolidation techniques and show that using inflation rates to guide consolidation can significantly reduce network utilization.

We observe that while both consolidation techniques perform comparable, there are situations when Slick consolidation outperforms CoMB's strict consolidation, reducing overall utilization by up to 50%. We examine the different element chains and observe that strict consolidation does not perform as well when chains contain a combination of elements with inflation factor > 0 and inflation factor < 0. In these cases, strict consolidation fails to account for the inflation factors and the resulting transformation in traffic that increase network utilization (*e.g.*, a decompressor/encryption element that increases the overall data transmitted).

7.6.2.1 General Scaling Properties

We evaluate Slick on scenarios that involve processing a different number of unique flow spaces and a random distribution of traffic sources and destinations.

Number of distinct flow spaces. Figure 47 shows the aggregate average bandwidth utilization for Slick placement and steering with varying number of flow spaces. We use a tree topology; in each run, we increase the number of flow spaces and applications and introduce an element chain in the network. We can see with increasing number of flow spaces Slick placement consistently performs within 10–15% of Optimal placement and outperforms Random placement for varying number of flow sizes. In all these experiment runs Optimal had 15 more copies of element instances than Random and Slick, corresponding to the number of switches in the network. Each flow space had four to eight sources and four to eight destinations in it but all the sources and destinations in each flow space were non-overlapping.

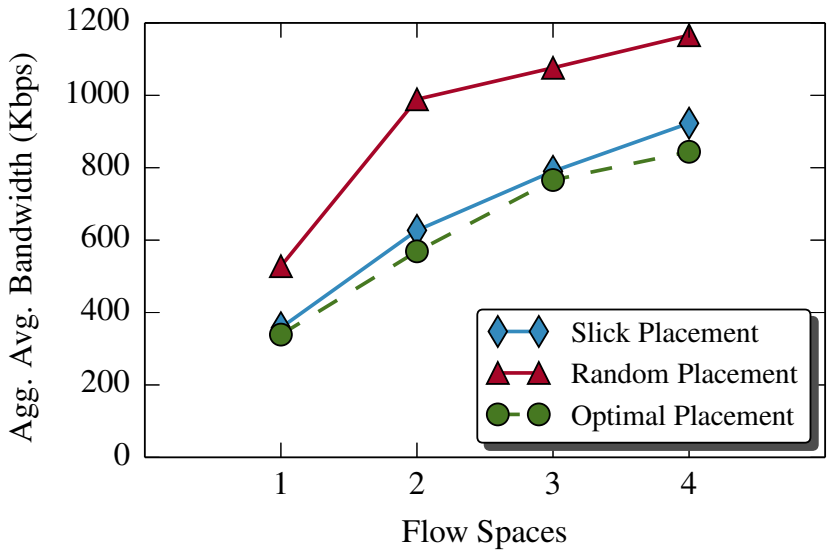
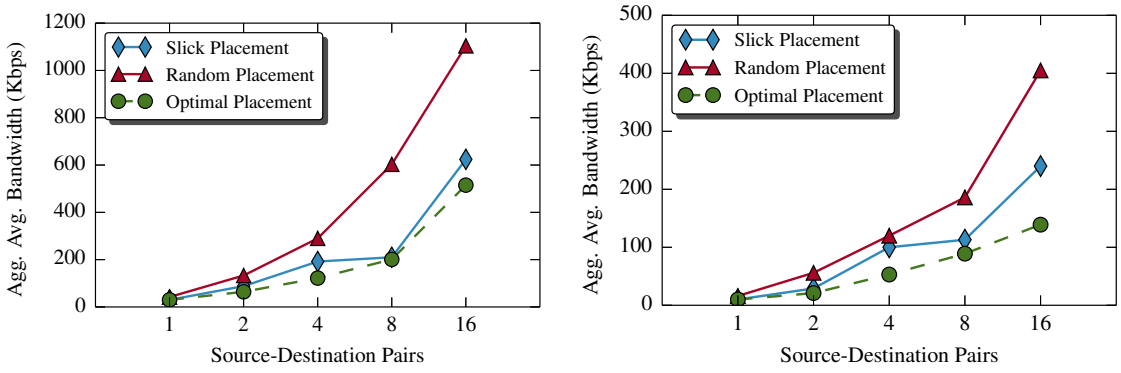


Figure 47: Slick placement performance with increasing number of unique flow spaces that the application processes.

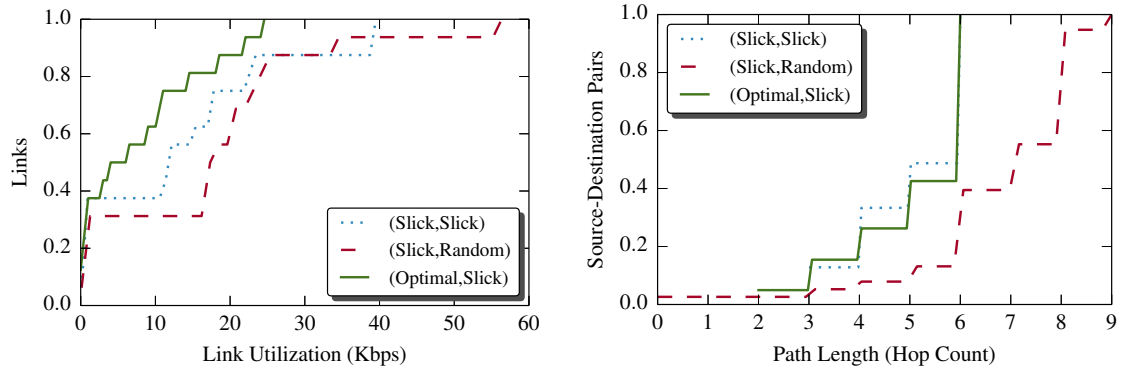
Increasing Source Destination Pairs. In above experiment the intersection of sources and destinations was an empty set for all the flow spaces but both East-West and North-South traffic pattern were present in them. In Figure 48, we present the aggregate average bandwidth utilization for the Slick algorithms with varying number of distributed switches all across the networks such that the intersection of source and destination switches can or cannot be an empty set. This experiment also has both North-South and East-West traffic flows. We use fixed tree and Fat-Tree topologies. Here we use a simple application with only one flow space. We deploy this application in both Tree and Fat-Tree topologies. For each experiment iteration, we randomly select source destination pairs and generate traffic between them. We increase the number of randomly selected host pairs from 1 to 16. As we can see that for both Tree 48a and Fat-Tree 48b topologies the Slick placement algorithm performance starts decreasing with increasing number of randomly distributed hosts. But for both topologies Slick placement consistently performs better than Random placement and in many cases performs comparably to Optimal.



(a) Tree Topology

(b) Fat-Tree Topology

Figure 48: Effect of different placement algorithms on traffic distribution, for different numbers of random source-destination pairs.



(a) Network Link Utilization.

(b) Path Lengths.

Figure 49: Quantifying the benefits of Slick's different algorithms.

7.6.2.2 Individual Placement and Steering Algorithms

We quantify the benefits of each of Slick's placement and steering algorithms by evaluating different combinations of placement and steering algorithms: (Slick,Random), a version of Slick with our placement algorithms but with Random steering; (Optimal, Slick) a version of Slick with our steering algorithm but the optimal placement; and (Slick,Slick) a version of Slick with Slick's placement and steering algorithms.

In Figure 49, we compare the link utilization and path lengths for the different algorithms. Figure 49 shows that Slick's steering algorithm contributes significantly to Slick's improvement's over random by providing reductions of both the median and

Table 13: Effect of network topology size on Slick's steering and placement algorithms.

Topology Size (Nodes)	Avg. Steering Time (ms)	Avg. Placement Time (ms)
15	13	131
31	15	404
63	11	581

99th percentile path lengths (19%,30%) and link utilization (37%,34%) over random steering. We used a Fat-Tree topology. We deploy one element to process traffic of one flow space; with 16 randomly distributed source destination switch pairs across the network. The traffic matrix has both East-West and North-South traffic flows. Figure 49a shows that Slick's placement results in higher link utilizations in exchange for deploying fewer element instances. Figure 49b shows that Slick placement places elements in locations that provide shorter path length by restricting the number of elements used. In real-world networks, the presence of background traffic may result in higher overall traffic latencies, but we expect the results to be qualitatively similar; the respective differences in network performance between different configurations may be larger, as a result of this increased background traffic.

7.6.3 Controller Performance

We now explore the scalability of Slick's control plane and examine the different parameters that can affect its performance. We evaluate how the following dimensions: *a*) Network Size; *b*) Size of elements in a chain; *c*) Number of Element Instances in each stage of the chain. affect the run times of Slick's placement and steering modules.

Network Size and Element Chain Size. To quantify the effects of network topology and element chain size on the Slick controller's performance, we run a Slick control application with multiple flow spaces and element chains on topologies of varying sizes. From Table 13, we observe that the time for placement is linear as a function of network size. Similarly, the placement algorithm's time as well as element

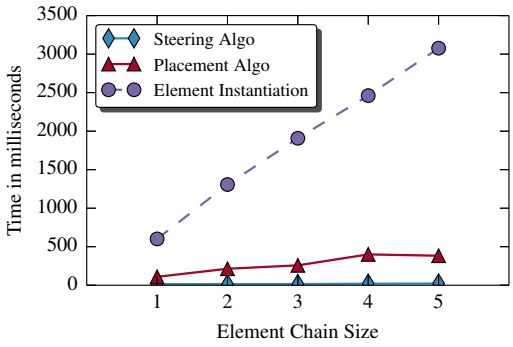


Figure 50: Effect of element chain size on Slick algorithms.

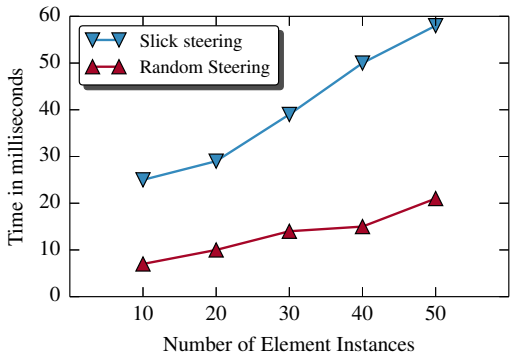


Figure 51: Performance of Slick's steering algorithm vs. random steering

instantiation time is impacted by number of elements in the element chain as shown in Figure 50. Topology size and element chain size both have a profound effect on the cost of placement. We also show that steering is understandably only impacted by the number of element instances, as we explain in more detail below.

Element Instances. In this experiment, we fix the network size and size of element chain and increase the number of element instances that can potentially operate on the flow space (*i.e.*, more element instances in each stage of Figure 43). As the number of element instances in the network increases, Slick's steering algorithm's computation time increases linearly, as shown in Figure 51. Since the steering algorithm will be called only on subset of flow spaces in a network (flows requiring Slick element processing), the longer time to run the algorithm is less of a concern. Additionally, Random steering in Figure 51 shows the lower bound for computation time for any steering algorithm implemented in Slick.

7.7 Summary

Most work on managing and orchestrating middleboxes has focused on deploying monolithic middleboxes, rather than deploying individual functions written in a high-level language. Slick takes the latter approach, representing a departure from existing designs, which provides the programmer with improved flexibility and scaling properties. Slick allows a programmer to write a single application that describes a sequence of processing elements for a given part of flow space, leaving the details of how those elements are replicated and placed throughout the network to the runtime. We presented a prototype and displayed the strength of the programming model by implementing several elements and realistic applications in previous chapter. In this chapter we have implemented different runtime algorithms that can be used to place the network elements across the network. We have also implemented multiple traffic steering algorithm to steer the traffic through network function enhancements while going from source to destination. We have showed that Slick's runtime can achieve near-optimal network bandwidth utilization for a variety of topologies and traffic loads.

CHAPTER 8

CONCLUDING REMARKS

In this thesis we have presented multiple techniques that can be used to enhance capabilities of network data planes using virtualization and software defined networking. The techniques presented here can be used to enhance capabilities of hardware chips, virtualize packet forwarding devices, improve packet forwarding switches, provide programming abstractions for the network functions and algorithms to place these functions and steer traffic through them with minimal impact on network utilization. In following section we summarize the contributions made and then discuss possible future directions for enhancement of network data plane capabilities.

8.1 Summary of Contributions

In chapter 3 we have displayed how virtualization on hardware chips can be used to provide support for multiple fast paths on a single chip with given hardware budget. We have demonstrated how a single FPGA can be programmed to host multiple data plane protocols side by side that can support both existing protocols and new protocols in network forwarding plane. Apart from hardware virtualization we have shown how custom software exceptions can be used to perform flexible packet functions in a virtualized hardware data plane.

In chapter 4 we have discussed how awareness of software virtualization can be exploited in hardware to support better performance for software based virtualized packet forwarding devices. In this work, we proposed, designed, implemented a proof of concept and evaluated it to see how virtualized network interface cards can be improved to reduce the housekeeping tasks of network I/O fairness for virtual machine hypervisors [25].

Software and hardware based approaches to virtualized data planes inside a network are used by different network equipment vendors to support data planes. In chapter 5 we design, implement and evaluate an architecture that can be used to add heterogeneous programming elements inside the network data plane. We show how a single device can provide both *forwarding plane* and *function plane* functionalities. We then present a runtime environment that can be used to program heterogeneous programming devices with a single programming interface from controller.

In first part of this thesis (chapters 3,4,5) we present techniques to enhance capabilities of virtualized packet forwarding devices. These enhancements are done at a single device level. Based on our experiences of these enhancements on packet forwarding devices we developed a programming abstraction that can be used to add custom packet processing functions on generalized packet processing machine in a given network. There are two main purposes of the programming abstraction 1) To ease the development of network functions for network function writers(Slick element interface). 2) To ease the deployment of existing network functions(Slick application interface).

The programming abstraction presented in chapter 6 shows how different packet processing functions can be written by network programmers and how they can be deployed by network operators. In chapter 7 we present a runtime that is used to implement the programming abstraction inside an actual network. We propose and implement a modular architecture that divides this runtime into three main modules. We then implement multiple algorithms for each of these modules. We show how different algorithms and heuristics can be used to reduce network resource utilization while implementing virtualized network functions on a physical network.

8.2 Future Directions

Introduction of new functions inside the data plane is an interesting research problem with its own set of challenges. With traditional packet forwarding devices main challenges were about packet forwarding speed, resource utilized to forward certain traffic volume, manageability of packet forwarding devices and complexity of the functions performed by these packet forwarding devices.

One of the most important question with programmable network functions is how network operators can program the network functions (both at the device level and at the network level) so that the performance, function complexity and resource utilization are at similar level as in the traditional packet forwarding devices/networks. Using different semiconductor technologies(FPGA, CPU etc.) combined with network virtualization and software defined networking this dissertation has shown how programmable data planes can be achieved while satisfying some of the standards(performance, flexibility and resource utilization) set forth by traditional packet forwarding networks.

Achieving performance, flexibility/complexity and resource utilization requirements set forth by traditional data planes is only part of the solution for programmable network functions. Introduction of new functions inside the data plane introduces various other challenges that need further investigation. Some of these challenges include but are not limited to security, reliability, stability, debugging suitability of the network enhancements, formal correctness guarantees and compatibility/interoperability (with existing infrastructure) of network enhancements. We think addressing these concerns will be a good future direction to achieve programmable data planes that can be used in real world deployments. Next we discuss how future work can address these concerns.

8.2.1 Debugging and Correctness

Implementation and deployment of correct data plane functions are two separate issues. A data plane function that is implemented correctly and passes all simulation and regression tests might not work or worse might not perform correctly in real deployment due to various issues. Apart from incorrect functioning of the new function itself, introduction/removal of a function can result in incorrect behavior for other functions part of the network function chain. Similarly, a network enhancement should not impact existing network policies.

An example of a new function enhancement causing problems with existing functions in the network and causing unintended consequences is following: IPv4 specification [123] and TRILL Protocol [121] require decrementing TTL(Time to Live) fields by Routers/RBridges [122] but in a network, a network function(other than Routers/RBridges) that modifies the TTL field for either of these protocols can result in reachability issues (loops and packets drops etc.) and higher bandwidth utilization. Similarly there is a need to quickly debug issues that can arise due to introduction or removal of such new functions.

In short, ensuring correctness and debugging of network functions is an important requirement and research work is needed that can enable confident addition or removal of functions for network data plane operations.

8.2.2 Security

Introduction or removal of new functions inside the data plane can result in unintended side effects. Apart from correctness issues mentioned earlier introducing new enhancements can result in network security problems. Data plane enhancements essentially mean adding new code to data plane functionality but it also means increasing the attack surface for the data planes. Traditional data plane development using standardization process and network equipment vendor is a long process but

it means that the standard specifications and developments go through a long peer reviewing process. It also implies that the data planes are implemented by network equipment vendors with years of design, development, testing and deployment experience. Despite this huge set of experience, many times the data plane implementations are exposed to security attacks [46, 77, 126]. There is a need for research that can ensure that introduction of new code does not introduce similar security loopholes. Apart from attacks on data planes, security loopholes in data plane function implementation can result in end host attacks. Research work that can ensure that new function code does not increase the potential attack surface and does not weaken the security of existing infrastructure would be an interesting future work direction.

8.2.3 Reliability and Stability

The traditional vendor based deployments of network equipment meant that correctness, stability and reliability of the network functions was provided by the vendor. But with recent works in custom data planes [22,32,33,43,56,79] and custom network functions [21,62,63,102,133] there is a need for the guarantee of correctness of these functions. Apart from correctness these enhancements need to be stable and reliable such that they can achieve five nines of reliability.

There has recently been work that has addressed the fault tolerance issues in software defined networks [44, 95] but most of this work looks at fault tolerance at the control plane level. One way to look at stability and reliability is to enable the data plane hardware and software design and implementation to be stable and reliable enough that it can provide five nines of availability required in computer and telecommunication networks.

8.2.4 Changing Network Traffic

Changing technology landscape combined with its new use cases results in new network applications and new ways of network traffic generation. Moreover, changes in

security [143] landscape have renewed emphasis in networking community to encrypt the data moving in the networks [73, 163].

With increased network traffic encryption there is a need for network functions that can be deployed inside the network without looking inside the packet contents. There has been recent research into doing deep packet inspection using custom encryption framework [138] but there is a need of research that can support different network functions(QoS(Quality of Service), DPI(Deep Packet Inspection), Prioritization etc.) with new and existing encryption algorithms and protocols.

Apart from encryption, new network applications and application level protocols keep generating new network traffic patterns. Programmable networks and the technology used to implement them also need to keep abreast of these applications and protocols in performance, resource utilization and complexity. In other words, new applications and changing network traffic will keep on providing interesting research and development problems for future programmable data planes.

REFERENCES

[1] "Accelize 40G Ethernet Card." http://www.accelize.com/products/fpga-platform-components/fpga-network-accelerator-cards/fpga-compute-card-xp7v690lp-40g.html.

[2] "AMD Graphics." http://www.amd.com/us/products/Pages/graphics.aspx.

[3] "Cavium Octeon Network Processors." http://www.cavium.com/OCTEON_MIPS64.html.

[4] "Cavium PCI-Express 10GbE Adapters." http://www.cavium.com/Intelligent_Network_Adapters_NIC10E.html.

[5] "GNU Debugger." http://www.gnu.org/software/gdb/.

[6] "Intel Atom Processor E6x5C Series." http://www.intel.com/p/en_US/embedded/hwsw/hardware/atom-e6x5c/overview.

[7] "Marvell Prestera 98CX8297." http://www.marvell.com//switching/assets/Marvell_Prestera_98CX8297-001_product_brief.pdf.

[8] "Marvell Xelerated HX Family of Network Processors." http://www.marvell.com/network-processors/xelerated-hx/.

[9] "Microsoft Hyper-V-Server." http://www.microsoft.com/hyper-v-server/en/us/default.aspx.

[10] "NetFPGA." http://www.netfpga.org.

[11] "Nvidia GPU." http://www.nvidia.com/page/products.html.

[12] "NVIDIA Management Library (NVML)." http://developer.nvidia.com/cuda/nvidia-management-library-nvml.

[13] "R7E-100 PCI Express Broadcom XLR Packet Processor Card." http://goo.gl/R8anu.

[14] "Intel's Teraflops Research Chip." http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf, 2010.

[15] "Product brief: Intel 82598 10gb ethernet controller." http://www3.intel.com/assets/pdf/prodbrief/317796.pdf, 2010.

[16] "Product brief: Neterion x3100 series." http://www.neterion.com/products/pdfs/X3100ProductBrief.pdf, 2010.

[17] "Latency: Not all numbers are measured the same," white paper, Juniper Networks, Inc., 2011.

[18] Al-Fares, M., Loukissas, A., and Vahdat, A., "A scalable, commodity, data center network architecture," in *ACM SIGCOMM Conference*, 2008.

[19] Andersen, D. G., Balakrishnan, H., Feamster, N., Koponen, T., Moon, D., and Shenker, S., "Accountable Internet Protocol (AIP)," in *Proc. ACM SIGCOMM*, (Seattle, WA), Aug. 2008.

[20] Anderson, J., Braud, R., Kapoor, R., Porter, G., and Vahdat, A., "xOMB: Extensible Open Middleboxes with Commodity Servers ," in *Proc. ANCS*, 2012.

[21] Anwer, B., Benson, T., Feamster, N., and Levin, D., "Programming slick network functions," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, p. 14, ACM, 2015.

[22] Anwer, B., Motiwala, M., bin Tariq, M., and Feamster, N., "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware," (New Delhi, India), Aug. 2010.

[23] Anwer, M. B. and Feamster, N., "Building a Fast, Virtualized Data Plane with Programmable Hardware," in *Proc. ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, (Barcelona, Spain), Aug. 2009.

[24] "Amazon web services." http://aws.amazon.com/.

[25] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the Art of Virtualization," in *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, (Lake George, NY), Oct. 2003.

[26] Bavier, A., Feamster, N., Huang, M., Peterson, L., and Rexford, J., "In VINI Veritas: Realistic and Controlled Network Experimentation," in *Proc. ACM SIGCOMM*, (Pisa, Italy), Aug. 2006.

[27] Benson, T., Akella, A., and Maltz, D., "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM Internet Measurement Conference*, (Melbourne, Australia), Nov. 2010.

[28] Benson, T., Akella, A., and Maltz, D. A., "Unraveling the complexity of network management.," in *NSDI*, pp. 335–348, 2009.

[29] Bhatia, S., Motiwala, M., Muhlbauer, W., Valancius, V., Bavier, A., Feamster, N., Peterson, L., and Rexford, J., "Hosting Virtual Networks on Commodity Hardware," Tech. Rep. GT-CS-07-10, Georgia Institute of Technology, Atlanta, GA, Oct. 2007.

[30] Bhatia, S., Motiwala, M., Mühlbauer, W., Valancius, V., Bavier, A., Feamster, N., Rexford, J., and Peterson, L., "Hosting virtual networks on commodity hardware," Tech. Rep. GT-CS-07-10, College of Computing, Georgia Tech, Oct. 2007.

[31] "Extreme networks: Blackdiamond x8 switch.." `http://www.extremenetworks.com/products/blackdiamond-x.aspx`.

[32] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and others, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[33] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 99–110, ACM, 2013.

[34] Bradner, S., *Benchmarking Terminology for Network Interconnection Devices*. Internet Engineering Task Force, July 1991. RFC 1242.

[35] Brebner, G., "Packets everywhere: The great opportunity for field programmable technology," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 1–10, IEEE, 2009.

[36] Brebner, G., James-Roxby, P., Keller, E., and Kulkarni, C., "Hyperprogrammable architectures for adaptable networked systems," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 328–338, IEEE, 2004.

[37] Caesar, M., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J., "Design and implementation of a routing control platform," in *Proc. 2nd USENIX NSDI*, (Boston, MA), May 2005.

[38] Calarco, G., Raffaelli, C., Schembra, G., and Tusa, G., "Comparative analysis of smp click scheduling techniques," in *QoS-IP*, pp. 379–389, 2005.

[39] Carli, L. D., Pan, Y., Kumar, A., Estan, C., and Sankaralingam., K., "Flexible lookup modules for rapid deployment of new protocols in high-speed routers," in *Proc. ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[40] Carpenter, B., *Middleboxes: Taxonomy and Issues*. Internet Engineering Task Force, Feb. 2002. RFC 3234.

[41] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S., "Ethane : Taking control of the enterprise," in *SIGCOMM '07*, 2007.

[42] Casado, M., Koponen, T., Moon, D., and Shenker, S., "Rethinking packet forwarding hardware," in *Proc. Seventh ACM SIGCOMM HotNets Workshop*, Nov. 2008.

[43] "XPliant Ethernet Switch Product Family." `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[44] Chandrasekaran, B. and Benson, T., "Tolerating SDN application failures with LegoSDN," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, p. 22, ACM, 2014.

[45] Charikar, M., Naamad, Y., Rexford, J., and Zou, K., "Multi-Commodity Flow with In-Network Processing," tech. rep., Princeton University, 2014. `http://www.cs.princeton.edu/~jrex/papers/mopt14.pdf`.

[46] Chasaki, D. and Wolf, T., "Attacks and defenses in the data plane of networks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 6, pp. 798–810, 2012.

[47] Cherkasova, L., Gupta, D., and Vahdat, A., "Comparison of the three cpu schedulers in xen," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, pp. 42–51, Sept. 2007.

[48] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler., D. R., "An empirical study of operating system errors," in *Symposium on Operating Systems Principles (SOSP)*, Dec. 2001.

[49] Chowdhury, N. M. K. and Boutaba, R., "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[50] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M., "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[51] Covington, G. A., Gibb, G., Lockwood, J., and McKeown, N., "A Packet Generator on the NetFPGA platform," in *FCCM '09: IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

[52] Covington, G. A., Gibb, G., Naous, J., Lockwood, J. W., and McKeown, N., "Encouraging reusable network hardware design," in *Microelectronic Systems Education, 2009. MSE'09. IEEE International Conference on*, pp. 29–32, IEEE, 2009.

[53] "Nvidia cuda." http://www.nvidia.com/object/cuda_home.html, Mar. 2011.

[54] Deering, S. and Hinden, R., *Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, Dec. 1998. RFC 2460.

[55] Dixon, C., Uppal, H., Brajkovic, V., Brandon, D., Anderson, T., and Krishnamurthy, A., "ETTM: A Scalable Fault Tolerant Network Manager," in *Proc. 8th USENIX NSDI*, (Boston, MA), Apr. 2011.

[56] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S., "RouteBricks: Exploiting parallelism to scale software routers," in *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, (Big Sky, MT), Oct. 2009.

[57] Doria, A., Haas, R., Salim, J., Khosravi, H., and Wang, W. M., *ForCES Protocol Specification*. Internet Engineering Task Force, Mar. 2007. Internet Draft (http://www.ietf.org/internet-drafts/draft-ietf-corces-protocol-09.txt). Work in progress, expires August 2007.

[58] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L., "Towards high performance virtual routers on commodity hardware," in *Proceedings of the 2008 ACM CoNEXT Conference*, p. 20, ACM, 2008.

[59] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Mathy, L., and Schooley, T., "Evaluating xen for router virtualization," in *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pp. 1256–1261, IEEE, 2007.

[60] "Enter the Andromeda zone - Google Cloud Platform latest networking stack." http://goo.gl/u59Iw1.

[61] "ETSI Network Function Virtualization." http://www.etsi.org/technologies-clusters/technologies/nfv.

[62] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C., "Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags," in *Proc. 11th USENIX NSDI*, (Seattle, WA), Apr. 2014.

[63] Fayazbakhsh, S. K., Sekar, V., Yu, M., and Mogul, J. C., "FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, (Hong Kong, China), Aug. 2013.

[64] Feamster, N., Balakrishnan, H., Rexford, J., Shaikh, A., and van der Merwe, K., "The case for separating routing from routers," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, (Portland, OR), Sept. 2004.

[65] Feamster, N., Rexford, J., and Zegura, E., "The Road to SDN: An Intellectual History of Programmable Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[66] Foster, N., Harrison, R., Freedman, M., Monsanto, C., Rexford, J., Story, A., and Walker, D., "Frenetic: A network programming language," in *International Conference on Functional Programming*, Sept. 2011.

[67] Franke, H., Xenidis, J., Basso, C., Bass, B. M., Woodward, S. S., Brown, J. D., and Johnson, C. L., "Introduction to the wire-speed processor and architecture," *IBM J. Res and Dev.*, vol. 54, Apr. 2010.

[68] Gember, A., Grandl, R., Anand, A., Benson, T., and Akella, A., "Stratos: Virtual Middleboxes as First-Class Entities," Tech. Rep. TR1771, University of Wisconsin-Madison, June 2012.

[69] Gember, A., Prabhu, P., Ghadiyali, Z., and Akella, A., "Toward software-defined middlebox networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 7–12, ACM, 2012.

[70] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A., "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM*, (Chicago, IL), pp. 163–174, ACM, 2014.

[71] Gibb, G., Covington, A., Yabe, T., and McKeown, N., "OpenPipes: Prototyping high-speed networking systems," Aug. 2009. SIGCOMM 2009 Demo Session.

[72] Godfrey, B., Ganichev, I., Shenker, S., and Stoica, I., "Pathlet routing," (Barcelona, Spain), Aug. 2009.

[73] "Google encrypts data amid backlash against NSA spying." `https://goo.gl/Jb3ZLu`.

[74] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H., "A clean slate 4D approach to network control and management," *ACM Computer Communications Review*, vol. 35, no. 5, pp. 41–54, 2005.

[75] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M., and Mathy, L., "Flow processing and the rise of commodity network hardware," *ACM Computer Communications Review*, Apr. 2009.

[76] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A., "Enforcing Performance Isolation Across Virtual Machines in Xen," in *Proc. of the USENIX 7th Intl. Middleware Conference*, Feb. 2007.

[77] Hadžić, I. and Smith, J. M., "Balancing performance and flexibility with hardware support for network architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 375–411, 2003.

[78] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S., "Network function virtualization: Challenges and opportunities for innovations," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, 2015.

[79] Han, S., Jang, K., Park, K., and Moon, S., "PacketShader: a GPU-accelerated software router," in *Proc. ACM SIGCOMM*, (New Delhi, India), Aug. 2010.

[80] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N., "Reproducible Network Experiments Using Container-based Emulation," in *Proc. ACM SIGCOMM CoNext Conference*, 2012.

[81] Heart, F. E., Kahn, R. E., Ornstein, S., Crowther, W., and Walden, D. C., "The interface message processor for the ARPA computer network," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, pp. 551–567, ACM, 1970.

[82] "40 Gb/s and 100 Gb/s Fiber Optic Task Force." `http://www.ieee802.org/3/bm/`.

[83] "400 Gb/s Ethernet Task Force." `http://www.ieee802.org/3/bs/`.

[84] "Internet Engineering Task Force." `http://ietf.org/`.

[85] "Intel IXP 2xxx Network Processors." `http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm`.

[86] Internet Engineering Task Force, *IP Network Address Translator (NAT) Terminology and Considerations*, Aug. 1999. RFC 2663.

[87] Jain, A., Hellerstein, J. M., Ratnasamy, S., and Wetherall, D., "A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers," in *Proc. 3nd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, (San Diego, CA), Nov. 2004.

[88] Jin, X., Li, E. L., Vanbever, L., and Rexford, J., "SoftCell: Scalable and flexible cellular core network architecture," in *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

[89] Joseph, D. and Stoica, I., "Modeling Middleboxes," *IEEE Network*, vol. 22, no. 5, pp. 20–25, 2008.

[90] Joseph, D. A., Tavakoli, A., and Stoica, I., "A Policy-aware Switching Layer for Data Centers," in *ACM SIGCOMM Conference*, 2008.

[91] Kim, C., Caesar, M., and Rexford, J., "Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises," (Seattle, WA), Aug. 2008.

[92] Kim, H., Benson, T., Akella, A., and Feamster, N., "Understanding the Evolution of Network Configuration: A Tale of Two Campuses," in *ACM SIGCOMM Internet Measurement Conference*, (Berlin, Germany), 2011.

[93] Kim, H. and Feamster, N., "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, 2013.

[94] Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R., "Kinetic: Verifiable dynamic network control," in *Proc. 12th USENIX NSDI*, (Oakland, CA), May 2015.

[95] Kim, H., Santos, J. R., Turner, Y., Schlansker, M., Tourrilhes, J., and Feamster, N., "Coronet: Fault tolerance for software defined networks," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pp. 1–2, IEEE, 2012.

[96] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F., "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, Aug. 2000.

[97] Labrecque, M., Steffan, J. G., Salmon, G., Ghobadi, M., and Ganjali, Y., "Netthreads: Programming netfpga with threaded software," in *NetFPGA Dev. Workshop*, 2009.

[98] Li, L. E., Liaghat, V., Zhao, H., Hajiaghayi, M., Li, D., Wilfong, G. T., Yang, Y. R., and Guo, C., "PACE: Policy-Aware Application Cloud Embedding.," in *IEEE INFOCOM*, (Turin, Italy), 2013.

[99] Liedtke, J., "On μ-kernel construction," in *Proc. of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[100] Linton, F., "A set of measures of centrality based upon betweenness," *Sociometry*, vol. 40, pp. 35–41, 1977.

[101] Lockwood, J. W., "Evolvable Internet hardware platforms," in *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*, pp. 271–279, IEEE, 2001.

[102] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y., "Serverswitch: a programmable and high performance platform for data center networks," in *Proc. 8th USENIX NSDI*, (Boston, MA), Apr. 2011.

[103] Lu, G., Miao, R., Xiong, Y., and Guo, C., "Using CPU as a Traffic Co-processing Unit in Commodity Switches," in *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, (Helsinki, Finland), Aug. 2012.

[104] Macedonia, M. R. and Brutzman, D. P., "Mbone provides audio and video across the internet," *Computer*, vol. 27, no. 4, pp. 30–36, 1994.

[105] "Magma PCI Expansion System." `http://www.magma.com/13slotrugged.asp`.

[106] Mansley, K., Law, G., Riddoch, D., Barzini, G., Turton, N., and Pope., S., "Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains," in *Euro-Par 2007 Workshops: Parallel Processing*, 2007.

[107] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F., "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (Seattle, WA), pp. 459–473, USENIX Association, Apr. 2014.

[108] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J., "OpenFlow: Enabling innovation in campus networks," *ACM Computer Communications Review*, Apr. 2008.

[109] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G., and Zwaenepoel, W., "Diagnosing performance overheads in the xen virtual machine environment," in *Conference on Virtual Execution Environments (VEE)*, June 2005.

[110] Monsanto, C., Foster, N., Harrison, R., and Walker, D., "A Compiler and Run-time System for Network Programming Languages ," in *ACM POPL*, (Philadelphia, USA), pp. 217–230, Jan. 2012.

[111] Motiwala, M., Elmore, M., Feamster, N., and Vempala, S., "Path Splicing," (Seattle, WA), Aug. 2008.

[112] Mysore, R. N., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A., "Portland: A Scalable Fault-Tolerant Layer2 Data Center Network Fabric," (Barcelona, Spain), Aug. 2009.

[113] Neely, C., Brebner, G., and Shang, W., "ShapeUp: A high-level design approach to simplify module interconnection on FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pp. 141–148, IEEE, 2010.

[114] "NOX: An OpenFlow controller." `http://www.noxrepo.org`.

[115] "Open networking foundation." `https://www.opennetworking.org/`.

[116] "The OpenCL Specification - Khronos Group." www.khronos.org/registry/cl/specs/opencl-1.1.pdf, Sept. 2012.

[117] "OpenFlow Switch Consortium." http://www.openflowswitch.org/, 2008.

[118] "OpenVZ: Server Virtualization Open Source Project." http://www.openvz.org.

[119] Partridge, C. and Others, "A 50-Gb/s IP router," *IEEE/ACM Transactions on Networking*, vol. 6, pp. 237–248, June 1998.

[120] Paxson, V., "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[121] Perlman, R., *"Transparent Interconnection of Lots of Links (TRILL): Problem and applicability statement"*. Internet Engineering Task Force, 2009. RFC 5556.

[122] Perlman, R., *"Routing Bridges (RBridges): Base Protocol Specification"*. Internet Engineering Task Force, July 2011. RFC 6325.

[123] Postel, J. B., *Internet Protocol*. Internet Engineering Task Force, Information Sciences Institute, Marina del Rey, CA, Sept. 1981. RFC 791.

[124] "POX: An OpenFlow controller." http://www.noxrepo.org/pox/about-pox/.

[125] Qazi, Z., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M., "SIMPLE-fying Middlebox Policy Enforcement using SDN," in *ACM SIGCOMM Conference*, 2013.

[126] Qian, Z. and Mao, Z. M., "Off-path tcp sequence number inference attack-how firewall middleboxes reduce security," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 347–361, IEEE, 2012.

[127] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A., "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *USENIX Symposium on Networked Systems Design and Implementation*, (Lombard, Illinois), 2013.

[128] Ram, K. K., Santos, J. R., Turner, Y., Cox, A. L., and Rixner, S., "Achieving 10 gb/s using safe and transparent network interface virtualization," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, Mar. 2009.

[129] "Remote direct memory access." http://www.rdmaconsortium.org/.

[130] Rixner, S., "Network virtualization: Breaking the performance barrier," *ACM Queue*, Jan. 2008.

[131] Rubow, E., McGeer, R., Mogul, J., and Vahdat, A., "Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware," in *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, pp. 1–10, IEEE, 2010.

[132] Santos, J. R., Turner, Y., Janakiraman, G. J., and Pratt, I., "Bridging the gap between software and hardware techniques for i/o virtualization," in *USENIX Annual Technical Conference*, June 2008.

[133] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G., "Design and implementation of a consolidated middlebox architecture," in *Proc. 9th USENIX NSDI*, (San Jose, CA), Apr. 2012.

[134] "Service Function Chaining Problem Statement(IETF Draft RFC)." `http://goo.gl/cQMV6k`.

[135] "SFC: Service Function Chaining." `https://datatracker.ietf.org/wg/sfc/charter/`.

[136] Shaikh, A. and Greenberg, A., "Experience in Black-box OSPF Measurement," in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, (San Francisco, California, USA), Nov. 2001.

[137] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V., "Making middleboxes someone elseś problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM*, (Helsinki, Finland), Aug. 2012.

[138] Sherry, J., Lan, C., Popa, R. A., and Ratnasamy, S., "Blindbox: Deep packet inspection over encrypted traffic,"

[139] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G., "Can the production network be the testbed?," in *Proc. 9th USENIX OSDI*, (Vancouver, Canada), Oct. 2010.

[140] Shieh, A., Kandula, S., and Sirer, E. G., "Sidecar: building programmable datacenter networks without programmable switches," in *Proc. ACM Hotnets-IX*, (Montery, CA. USA.), Oct. 2010.

[141] Shin, S., Porras, P. A., Yegneswaran, V., Fong, M. W., Gu, G., and Tyson, M., "Fresco: Modular composable security services for software-defined networks.," in *Proc. NDSS*, 2013.

[142] "Snort intrusion detection system." `https://www.snort.org/`.

[143] "Leaked Snowden Documents Show Expansion of Cybersurveillance by U.S. Agencies." `http://goo.gl/ZBsFmA`.

[144] Soule, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., G.Sirer, E., and Foster., N., "Merlin:a language for provisioning network resources," in *Proc. CoNEXT*, Dec. 2014.

[145] Srisuresh, P. and Egevang, K., *Forwarding and Control Element Separation (ForCES) Framework*. Internet Engineering Task Force, Jan. 2001. RFC 3022.

[146] Swift, M. M., Bershad, B. N., and Levy, H. M., "Improving the reliability of commodity operating systems," *ACM Transactions on Computer Systems*, pp. 77–100, 2005.

[147] Taylor, D. E., Turner, J. S., Lockwood, J. W., and Horta, E. L., "Dynamic hardware plugins: exploiting reconfigurable hardware for high-performance programmable routers," *Computer Networks*, vol. 38, no. 3, pp. 295–310, 2002.

[148] Turner, J., Crowley, P., DeHart, J., Freestone, A., Heller, B., Kuhns, F., Kumar, S., Lockwood, J., Lu, J., Wilson, M., and others, "Supercharging planetlab: a high performance, multi-application, overlay network platform," Aug. 2007.

[149] Unnikrishnan, D., Lu, J., Gao, L., and Tessier, R., "Reclick-a modular dataplane design framework for fpga-based network virtualization," in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pp. 145–155, IEEE Computer Society, 2011.

[150] Unnikrishnan, D., Vadlamani, R., Liao, Y., Dwaraki, A., Crenne, J., Gao, L., and Tessier, R., "Scalable network virtualization using fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 219–228, ACM, 2010.

[151] Van der Merwe, J. E., Rooney, S., Leslie, I., and Crosby, S., "The tempest-a practical framework for network programmability," *Network, IEEE*, vol. 12, no. 3, pp. 20–28, 1998.

[152] van der Merwe, J., Rooney, S., Leslie, I., and Crosby, S., "The Tempest - A Practical Framework for Network Programmability," *IEEE Network*, vol. 12, pp. 20–28, May 1998.

[153] "Virtual interface architecture." `http://www.cs.uml.edu/~bill/cs560/VI_spec.pdf`.

[154] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P., "Maple:simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM Conference*, 2013.

[155] "Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks." `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-01`.

[156] Wang, Z., Qian, Z., Xu, Q., Mao, Z., and Zhang, M., "An untold story of middleboxes in cellular networks," in *Proc. ACM SIGCOMM*, (Toronto, Ontario, Canada), Aug. 2011.

[157] Wolf, T. and Turner, J. S., "Design issues for high-performance active routers," *Selected Areas in Communications, IEEE Journal on*, vol. 19, no. 3, pp. 404–409, 2001.

[158] Xilinx, "Virtex-7 series." http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html.

[159] Xilinx, "Virtex-ii pro." http://www.xilinx.com/support/documentation/virtex-ii_pro.htm.

[160] Xilinx, "Virtex-ultrascale." http://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html.

[161] Xilinx, "Xilinx ise design suite." http://www.xilinx.com/tools/designtools.htm.

[162] Xilinx, "Virtex-4 overview." http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/overview/index.htm, Feb. 2006.

[163] "Yahoo now encrypting traffic from its data centers, and plans to encrypt Messenger too." https://goo.gl/GnG35F.

[164] Yang, L., Dantu, R., Anderson, T., and Gopal, R., *Forwarding and Control Element Separation (ForCES) Framework*. Internet Engineering Task Force, Apr. 2004. RFC 3746.

[165] Yang, X., Wetherall, D., and Anderson, T., "Source selectable path diversity via routing deflections," (Pisa, Italy), Aug. 2006.

[166] Zhang, Y., Beheshti, N., Beliveau, L., Lefebvre, G., Misra, R., Patney, R., Rubow, E., Subrahmaniam, R., Manghirmalani, R., Shirazipour, M., Truchan, C., and Tatipamula, M., "StEERING: A Software-Defined Networking for Inline Service Chaining," in *IEEE ICNP*, (Goettingen, Germany), IEEE, 2013.